AFRL-RV-PS-TR-2007-1209 **STINFO COPY** **AFRL-RV-PS-TR-2007-1209**

Polymorphous Computing Architectures

Mark Horowitz

Stanford University
Gates 306, 353 Serra Mall
Stanford, CA 94305-9030

12 December 2007

Final Report

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
3550 Aberdeen Ave SE
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

NOTICE AND SIGNATURE PAGE

//signed//	//signed//
JEFFREY B. SCOTT, 2d Lt USAF Program Manager	JOHN P. BEAUCHEMIN, Lt Col, USAF Deputy Chief, Spacecraft Technology Division Space Vehicles Directorate

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 12/12/2007	**2. REPORT TYPE** Final Report	**3. DATES COVERED** *(From - To)* 13/03/2003 to 12/12/2007
4. TITLE AND SUBTITLE Polymorphous Computing Architectures		**5a. CONTRACT NUMBER** F29601-03-2-0117
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 62712E; 62716E
6. AUTHOR(S) Mark Horowitz		**5d. PROJECT NUMBER** ARPA
		5e. TASK NUMBER SC
		5f. WORK UNIT NUMBER AC
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Stanford University Gates 306, 353 Serra Mall Stanford, CA 94305-9030		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Space Vehicles Directorate 3550 Aberdeen Ave., SE Kirtland AFB, NM 87117-5776		**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RVSE
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RV-PS-TR-2007-1209

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited. (Clearance #RV08-0006).

13. SUPPLEMENTARY NOTES
.
.

14. ABSTRACT

We describe the architecture and hardware implementation of a coarse grain parallel computing system with flexibility in both memory and processing elements. The memory subsystem supports a wide range of programming models efficiently, including cache coherency, message passing, streaming, and transactions. The memory controller implements these models using metadata stored with each memory block. Processor flexibility is provided using Tensilica Xtensa cores. We use Xtensa processor options and Tensilica Instruction Extension language (TIE) to provide additional computational capabilities, to define additional memory operations needed to support our controller, and to add VLIW instructions for increased efficiency. In our implementation, two processors share multiple memory blocks via a load/store unit and a crossbar switch. These dual processor tiles are grouped into quads that share a memory protocol controller. Quads connect to one another and to the off-chip memory controller via a mesh-like network. We describe the design of each block in detail. We also describe our implementation of transactional memory. Transactional Coherence and Consistency (TCC) provides greater scalability than previous TM architectures by deferring conflict detection until commit time and by using directories to reduce overhead. We demonstrate near linear scaling up to 64 processors with less than 5% overhead.

15. SUBJECT TERMS
Polymorphic, Reconfigurable, Parallel Programming, Thread Programming, Stream Programming

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** 2d Lt Jeffrey B. Scott
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	Unlimited	95	**19b. TELEPHONE NUMBER** *(include area code)* (505) 846-6280

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

Table of Contents

Figures

1 Introduction

For a long time microprocessor designers focused on improving performance of sequential applications on single processor machines, achieving an annual performance growth rate of over 50% [1]. This phenomenal performance growth relied on three main factors: exploiting instruction-level parallelism (ILP), decreasing the number of “gates” in each clock cycle by building faster functional units and longer instruction pipelines, and using the faster transistors provided by CMOS technology scaling [2, 3]. Unfortunately, the first two factors have reached their limit; as a result of this and limitations such as wire delay and slowly changing memory latency, single processor performance growth has slowed down dramatically [1-3]. In addition, increasing complexity and deeper pipelining reduce the power efficiency of high-end microprocessors [4, 5]. These trends led researchers and industry towards parallel systems on a chip [6-16]. Parallel systems can efficiently exploit the growing number of transistors provided by continued technology scaling [2].

Programmers must re-write application software to realize the benefits of parallel systems on a chip. Since traditional parallel programming models such as shared memory and message-passing are not easy to use, researchers have proposed a number of new programming models. Two of the most popular today are *streaming* [17-19] and *transactions* [20, 21]. Although these new programming models are effective for some applications, they are not universal and the traditional shared memory model is still being used, especially in conjunction with a thread package. Also, new programming models are still evolving as researchers refine their APIs [22-24].

The goal of the Stanford Smart Memories project is to design a flexible architecture that can support several programming models and a wide range of applications. Since processors are fundamentally flexible – their operation is set by the code they run, our focus was on making the memory system as flexible as the processors. Our approach is to design a coarse-grain architecture that uses reconfigurable memory blocks [25] and a programmable protocol controller to provide the flexible memory system. Memory blocks have additional meta-data bits and can be configured to work as various memory structures, such as cache memory or local scratchpad, as necessary for a particular programming model or application. The protocol controller can be programmed to support different memory protocols, like cache coherence or transactional coherence and consistency (TCC) [21].

The Smart Memories architecture allows researchers to compare different programming models and memory system types under the same hardware resource constraints and to experiment with hybrid programming models [26, 27].

The rest of this section briefly reviews the memory and programming models we have already implemented on this architecture and related hardware required for their implementation. These programming models informed our reasoning about what were the required types of memory operations. The bulk of the final report is in Section 2, which describes the resulting hardware implementation in more detail. The design of this hardware was the primary output of this research effort. This architecture will be taped-out in Jan 2008, and we hope to have a system up and running later in that year.

In addition to creating the Smart Memory Architecture, this research effort also pushed forward the Transactional Memory programming model. This work is described in more detail in Section 3.

1.1 Cache-Coherent Shared Memory Model

In cache-coherent shared memory systems, only off-chip DRAM memory is directly addressable by all processors. Because off-chip memory is slow compared to the processor, fast on-chip cache memories are used to store the most frequently used data and to reduce the average access latency. Cache management is performed by hardware and does not require software intervention. As a processor performs loads and stores, hardware attempts to capture the working set of the application by exploiting *spatial and temporal locality*. If the data requested by the processor is not in the cache, the controller replaces the cache line least likely to be used in the future with the appropriate data block fetched from DRAM.

Software threads running on different processors communicate with each other implicitly by writing and reading shared memory. Since several caches can have copies of the same cache line, hardware must guarantee *cache coherence*, i.e. all copies of the cache line must be exactly the same. Hardware implementations of cache coherence typically follow an invalidation protocol: a processor is only allowed to modify an exclusive copy of the cache line, and all other copies must be invalidated before the write. Invalidation is performed by sending read for ownership requests to other caches. A common optimization is to use cache coherence protocols such as MESI (Modified/Exclusive/Shared/Invalid), which reduce the number of cases where remote cache lookups are necessary.

To resolve races between processors for the same cache line, requests must be *serialized*. In small scale shared memory systems serialization is performed by a shared bus which broadcasts every cache miss request to all processors. The processor which wins bus arbitration receives the requested cache line first. Bus-based cache coherent systems are called also *symmetric multi-processors* (SMP) because any main memory location can be accessed by any processor with the same average latency.

High latency and increased contention make the bus a bottleneck for large multiprocessor systems. *Distributed shared memory* (DSM) systems eliminate this bottleneck by physically distributing both processors and memories, which then communicate via an interconnection network. Coherence serialization is performed by directories associated with DRAM memory blocks. Directory-based cache coherence protocols try to minimize communication by keeping track of cache line sharing in the directories and sending invalidation requests only to processors which previously requested the cache line. DSM systems are also called *non-uniform memory access* (NUMA) architectures because average access latency depends on processor and memory location. Development of high-performance applications for NUMA systems can be significantly more complicated because programmers need to pay attention to where the data is located and where the computation is performed.

Chip multiprocessors (CMP) have significantly higher interconnect bandwidth and lower communication latencies than traditional multi-chip multiprocessors. This implies that the efficient design points for CMPs are likely to be different from those for traditional SMP and DSM systems. Also, even applications with non-trivial amount of data sharing and communication can perform and scale reasonably well. At the same time, modern CMPs are often limited by total power dissipation; low power is consequently one of the main goals of cache coherence design.

To improve performance and increase concurrency, multiprocessor systems try to overlap and re-order cache miss refills. This raises the question of a *memory consistency model*: what event ordering does hardware guarantee [28]. *Sequential consistency* guarantees that accesses from each individual processor appear in program order, and that the result of execution is the same as if all accesses from all processors were executed in some sequential order. *Relaxed consistency* models give hardware more freedom to re-order memory operations but require programmers to annotate application code with synchronization or memory barrier instructions to insure proper memory access ordering.

To synchronize execution of parallel threads and to avoid *data races* programmers use synchronization primitives such as *locks and barriers*. Implementation of locks and barriers requires support for *atomic read-modify-write* operations, e.g. compare-and-swap or load-linked/store-conditional. Parallel application programming interfaces (API) such as POSIX threads [29] and ANL macros [30] define application level synchronization primitives directly used by the programmers in the code.

1.2 Streaming Memory Model

In streaming architectures fast on-chip storage is organized as directly addressable memories called *scratchpads*, *local stores*, or *stream register files* [9, 13, 17]. We use the term scratchpad in this report. Data movement within chip and between scratchpads and off-chip memory is performed by direct memory access (DMA) engines which are directly controlled by application software. As a result software is responsible for managing and optimizing all aspects of communication: location, granularity, allocation and replacement policies, and the number of copies. For applications with simple and predictable data flow all data communication can be scheduled in advance and completely overlapped with computation, thus hiding communication latency.

Since data movements are managed explicitly by software, complicated hardware for coherence and consistency is not necessary. Hardware architecture should support DMA transfers between local scratchpads and off-chip memory.[1] Processors can access their local scratchpads as FIFO queues or as randomly indexed memories [31].

Streaming is similar to message-passing applied in the context of CMP design. However, there are several important differences from traditional message-passing in clusters and massively parallel systems. Communication is managed at the user level software and its overhead is low. Messages are exchanged at the first level of memory hierarchy, not the last one, and software has to take into account limited size of local scratchpads. Since communication between processors happens within a chip, the latency is low and the bandwidth is high. Finally, software manages both the communication between processors and the communication between processor scratchpads and off-chip memory.

Researchers have proposed several stream programming languages: StreamC/KernelC [17], StreamIt [19], Brook GPU [32], and Sequoia [33]. These languages differ in the level of abstraction but they share some basic concepts. Streaming computation must be divided into a set of *kernels*, i.e. functions which can not access arbitrary global state. Inputs and outputs of the kernel are called *streams* and must be specified explicitly as kernel arguments. Stream access patterns are typically restricted. Another important concept is *reduction variables* which allow a kernel to do calculations involving all elements of the input stream, such as the stream's summation.

Restrictions on data usage in kernels allow streaming compilers to determine computation and input data per element of the output stream, to parallelize kernels across multiple processing elements, and to schedule all data movements explicitly. In addition, the compiler optimizes the streaming application by splitting or merging kernels to balance loading, to fit all required kernel data into local scratchpads, or to minimize data communication through *producer-consumer locality* [17]. The complier also tries to overlap computation and communication by performing *stream scheduling*: DMA transfers run during kernel computation, which is equivalent to macroscopic prefetching.

To develop a common streaming compiler infrastructure, Stanford researchers have proposed the *stream virtual machine* (SVM) abstraction [34, 35]. SVM gives

[1]Some recent stream machines use caches for the control processor. In these cases, while the local memory does not need to maintain coherence with the memory, the DMA often needs to be consistent with the control processor. Thus in the IBM Cell processor the DMA engines are connected to coherent bus and all DMA transfers are performed to coherent address space [13].

high-level optimizing compilers for stream languages a common intermediate representation.

1.3 Transactional Memory Model

The traditional shared memory programming model usually requires programmers to use low-level primitives such as locks for thread synchronization. Locks are required to guarantee mutual exclusion when multiple threads access shared data. However, locks are hard to use and error-prone – especially when programmer is trying to avoid *deadlock* or to improve performance and scalability by using *fine-grain locking* [36]. Lock-based parallel applications can also suffer from *priority inversion*, and *convoying* [20]. These arise when subtle interaction between locks cause high priority tasks to wait for lower priority tasks to complete.

Lock-free shared data structures allow programmers to avoid problems associated with locks [37]. This methodology requires only standard compare-and-swap instruction but introduces significant overheads and thus it is not widely used in practice.

Transactional memory was proposed as a new multiprocessor architecture and programming model intended to make lock-free synchronization as efficient as conventional techniques based on locks [20]. Transactional memory allows programmers to define custom read-modify-write operations that can be applied to multiple arbitrary words in memory. The programmer must annotate applications with start transaction/end transaction commands; the hardware executes all instructions between these commands as a single atomic operation. Other processors or threads can only observe transaction state before or after execution; intermediate state is hidden. If transaction conflict is detected, such as one transaction updating a memory word read by another transaction, one of conflicting transactions must re-executed.

The concept of transactions is the same as in database management systems (DBMS). In DBMS, transactions provide the properties of atomicity, consistency, isolation, and durability (ACID). Transactional memory programming model is similar to database programming. The key difference is the number of instructions per transaction and the amount of state read or written by the transaction.

Transactional memory implementations have to keep track of transaction *read-set*, all memory words read by the transaction and *write-set*, and all memory words written by the transaction. Read-set is used for conflict detection between transactions, while write-set is used to track speculative transaction changes which will become visible after transaction *commit* or will be dropped after transaction *abort*. Conflict detection can be either *eager* or *lazy*. Eager conflict detection checks every individual read and write performed by the transaction to see if there is a collision with another transaction. Such an approach allows early conflict detection but requires read and write sets to be visible to all other transactions in the system. In the lazy approach, conflict detection is postponed until the transaction tries to commit.

Another design choice for transactional memory implementations is the type of version management. In *eager version management*, the controller writes speculative data directly into the memory as a transaction executes and keeps an *undo log* of the old values [38]. Eager conflict detection must be used to guarantee transaction atomicity with respect to other transactions. Transaction commits are fast since all data is already in place but aborts are slow because old data must be copied from the undo log. This approach is preferable if aborts are rare but may introduce subtle complications such as *weak atomicity* [39]: since transaction writes change the architectural state of the main memory they might be visible to other threads that are executing non-transactional code.

Lazy version management is another alternative, where the controller keeps speculative writes in a separate structure until a transaction commits. In this case aborts are fast since the state of the memory is not changed but the commits require more work. It is easier to support *strong atomicity*: complete transaction *isolation* from both transactions and non-transactional code executed by other threads [39].

Transactional memory implementations can be classified as hardware approaches (HTM) [20, 21], software-only (STM) techniques [40], or mixed approaches. Two mixed approaches have been proposed: hybrid transactional memory (HyTM) supports transactional execution in hardware but falls back to software when hardware resources are exceeded [41, 42], while hardware-assisted STM (HaSTM) combines STM with hardware support to accelerate STM implementations [44, 45].

In some proposed hardware transactional memory implementations, a separate transactional or conventional data cache is used to keep track of transactional reads and writes [20]. For either cache type, transactional support extends existing coherence protocols such as MESI to detect collisions and enforce transaction atomicity. The key issues with such approaches are arbitration between conflicting transactions and dealing with overflow of hardware structures. Memory consistency is also an issue since application threads can execute both transactional and non-transactional code.

Transactional coherence and consistency (TCC) is a transactional memory model in which atomic transactions are always the basic unit of parallel work, communication, and memory coherence and consistency [21]. Each of the parallel processors in TCC model continually executes transactions. Each transaction commits its writes to shared memory only as an atomic block after arbitration for commit. Only one processor can commit at a time by broadcasting its transactional writes to all other processors and to main memory. Other processors check incoming commit information for read-write dependency violations and restart their transactions if violations are detected. Instead of imposing some order between individual memory accesses, TCC serializes transaction commits. All accesses from an earlier committed transaction appear to happen before any memory references from a later committing transaction, even if actual execution was performed in an interleaved fashion. The TCC model guarantees strong atomicity because the TCC application only consists of transactions. Hardware overflow is also easy to handle: a transaction that detects overflow before commit stalls, and must arbitrate for the commit token. Once it has the token, it is no longer speculative, and can commit its previously speculative changes to free up hardware resources, and then continue execution. It can't release the commit token until it hits commit point in the application. Clearly this serializes execution, since only one thread can have the commit token at a time, but it does allow overflows to be cleanly handled.

A Programmer using TCC divides the application into transactions that will be executed concurrently on different processors. The order of transaction commits can be optionally specified. Such situations usually correspond to different phases of the application which must be separated by synchronization barriers in lock-based model. To deal with such ordering requirements TCC has hardware-managed *phase numbers* for each processor which can be optionally incremented upon transaction commit. Only transactions with oldest phase number are allowed to commit at any time.

Stanford researchers have proposed the OpenTM application programming interface (API), which provides a common programming interface for various transactional memory architectures [24].

1.4 Memory System Features

In looking over all three programming models, we find a number of common elements. All the designs use fast local memory close to the processor. This memory often has some state bits associated with it, but the use of the state is different in the different programming models. For example, in a cache it tracks the "state" of the cache line, for transactional memory it needs also to track data that is speculatively read or written, and for streaming it might be used to track whether the DMA has actually fetched a given word. In addition to the different uses of the state bits, the amount and connection of the local memories also differs in different machine models, and even between different applications within a programming model. This meant we needed to provide both flexible state bits in our memory system, and method for the processor to leverage these bits. The design of this flexible memory system is described in Section 2.3.

In addition to the flexible local memory, all these memory systems require a relatively sophisticated protocol engine that can handle the movement of data between the main memory and the local memories. Each of the programming models thinks about this movement differently, DMAs for streaming, cache fills and spills for shared memory, and commits in transactions, but the underlying operations are all pretty similar. They all require a mechanism to move blocks of data between the cache and the memory system, track outstanding requests, and for some protocols, maintain some ordering of events. In the Smart Memory system, the reconfigurable protocol controller handles these functions, and is described in Section 2.5.

The core processor that connects to the memory system initially was a custom designed processor that could flexible change its resources to better match the computational requirements of the different programming models. During the start of this research effort we realized that we could build this processor, but we did not have the resources to create the complete software tool chain that would be needed to make the processor truly useful. At that point we decided to use the Tensilica configurable processor, and use the configuration to make it work in our system. The next section reviews the overall design of our Smart Memory architecture, beginning with the overview of the entire system and a description of the memory clustering that was used. Section 2.2 describes the processor core in more detail, including how we used the Tensilica system to provide the special memory operations we needed.

2 Smart Memories Implementation

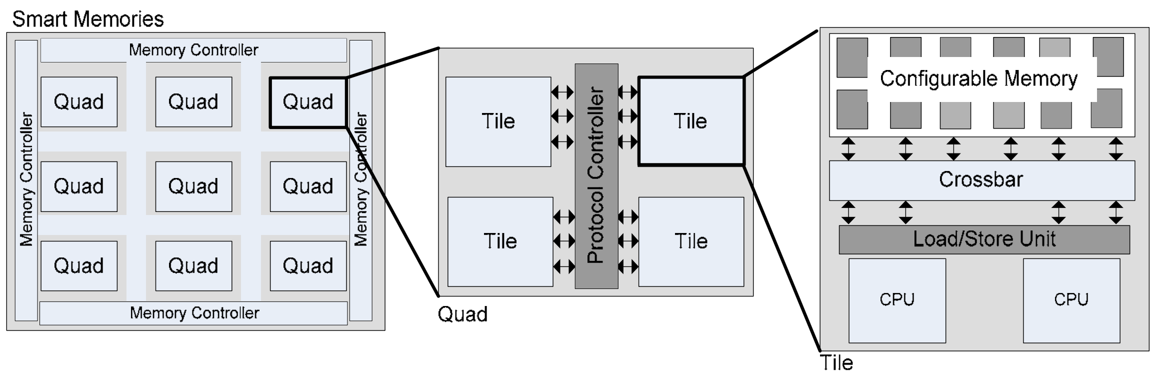


Figure 1 Smart Memories Architecture

The Smart Memories architecture was designed to support three different programming models and to accommodate VLSI physical constraints. As shown in Figure 1, the system consists of Tiles, each with two VLIW cores, several reconfigurable memory blocks, and a crossbar connecting them. Four adjacent Tiles form a Quad. Tiles in the Quad are connected to a shared protocol controller. Quads are connected to each other and to the off-chip interfaces using a mesh-like network.

The reconfigurable memory system is the key element of the Smart Memories architecture that allows it to support different programming models. The design of the memory system is based on the observation that, although different programming models place different requirements on the memory systems, the underlying hardware resources and operations are very similar. For example, the same memory blocks can be used to store data in caches or in stream register files; extra memory bits are used to store meta-data such as cache line state in conventional caches and “speculative” bits in transactional caches [21]. By adding a small amount of extra logic we can configure the same memory resources to behave differently.

The memory system consists of three major reconfigurable blocks, highlighted in Figure 1. The memory interface in each Tile (Load/Store Unit) coordinates accesses from processor cores to local memories and allows reconfiguration of basic memory accesses. A basic operation, such as a Store instruction, can treat a memory word differently in transactional mode than in conventional cache coherent mode. The memory interface can also broadcast accesses to a set of local memory blocks. For example, when accessing a set-associative cache, the access request is concurrently sent to all the blocks forming the cache ways.

Each memory mat in the Tile is an array of data words; each data word is associated with a few meta-data bits. Meta-data bits store the status of that data word and their state is considered in every memory access; an access to this word can be discarded based on the status of these bits. For example, when mats are configured as a cache, these bits are used to store the cache line state, and access is discarded if the status indicates that cache line is invalid. The meta data bits are dual ported, so they are updated atomically with access to the data word. The update functions are set by the configuration. A built-in comparator and a set of pointers allow the mat to be used as tag storage (for cache) or as a FIFO. Mats are connected to each other through an inter-mat network that communicates control information when the mats are accessed as a group. While the hardware cost of reconfigurable memory blocks is high in our standard-cell prototype, full custom design of such memory blocks can be quite efficient [25].

The protocol controller is a reconfigurable control engine that can execute a sequence of basic memory system operations to support memory mats. These operations include loading and storing data words (or cache lines) into mats, manipulating meta-data bits, keeping track of outstanding requests from each Tile, and broadcasting data or control information to Tiles within the Quad. This controller is connected to a network interface for sending and receiving requests to/from other Quads or off-chip interfaces.

Mapping a programming model to the Smart Memories architecture requires configuration of memory mats, Tile interconnect and protocol controller. For example, when implementing a shared-memory model, memory mats are configured as instruction and data caches, the Tile crossbar routes processor instruction fetches, loads, and stores to the appropriate memory mats, and the protocol controller acts as a cache coherence engine, which refills the caches and enforces coherence invariance.

2.1 Memory System Protocols

As previously mentioned, Smart Memories supports both a cache-coherent shared memory model and Transactional Coherence and Consistency. This section describes memory system protocols to implement necessary semantics of these programming models. In addition, a fine-grain synchronization protocol, which is used by all of the target programming models for providing synchronization primitives, is described in this section.

2.1.1 Coherence protocol

In the shared memory model, the memory system provides processors with local instruction and data cached to capture spatial and temporal locality of the applications' memory accesses. These caches need to be kept coherent by hardware such that each read to a memory address returns the value of the last write.

Smart Memories implements a hierarchical MESI coherence protocol to enforce coherence between all caches in the system. Data and instruction address spaces are kept coherent separately and not against one another. Coherence is maintained in two steps: a protocol controller in the Quad keeps caches within the Quad coherent, while one or more memory controllers are responsible for enforcing coherence among Quads. Memory requests go through points of serialization; memory requests from different caches that are within a Quad are serialized in the Quad's protocol controller, while memory requests from different Quads are serialized at the global level by the home memory controller for the specific cache line.

After a cache miss request passes the serialization point in the Quad's protocol controller, other Quad caches are searched (or "snooped") to update the state of the cache line according the MESI protocol. The protocol controller is capable of performing cache-to-cache transfers between Quad caches and merging requests from different sources inside the Quad to reduce latency and bandwidth requirements. If a cache request cannot be satisfied locally, a miss request is sent to the home memory controller to fetch the cache line.

The home memory controller of the cache line essentially takes the same steps at a higher level; after receiving a cache miss request and serializing it against any other outstanding cache misses, it broadcasts appropriate coherence requests to all Quads except the one which sent the original cache miss to inquire about the state of the cache line. Protocol controllers receive coherence requests, snoop caches and update the state according to MESI protocol. They send coherence replies back to the inquiring memory controller which might or might not contain the cache line (depending on type of the request and state of the cache line within the Quad).

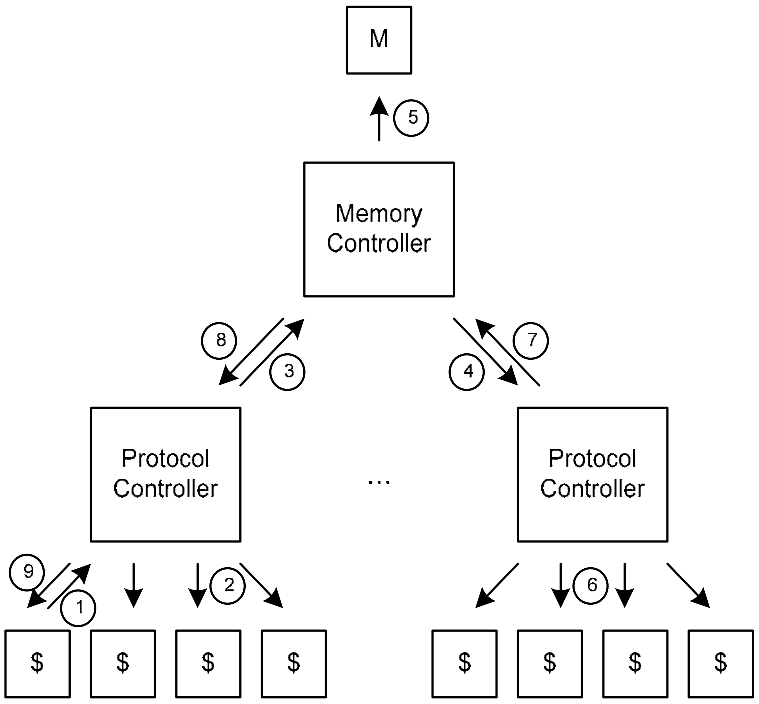


Figure 2 Steps in enforcing coherence

Figure 2 shows the steps involved in processing a sample cache miss request. These steps are as follows:

1- Cache miss request is received by protocol controller and is serialized properly
2- Local caches are snooped, state of any local copy of the cache line is adjusted
3- Cache miss request is sent to memory controller and is serialized properly with respect to requests from other Quads
4- Coherence requests are sent to Quads except the one that originated cache miss
5- Main memory is accessed in parallel go fetch the cache line
6- Protocol controller snoops caches and adjusts cache line states
7- Coherence replies are sent back to memory controller, which may or may not contain the actual cache line
8- Reply for the original cache miss is sent back after collecting all replies
9- Cache line is refilled and requested data word is sent to processor

Protocol controllers are capable of merging requests from different processors within the Quad to avoid sending unnecessary requests to memory controllers. Also, coherence can be turned on or off for parts of the address space by setting control bits within the protocol and memory controllers. This allows the private regions of the address space to be marked as not coherent and hence reduces the coherence traffic as well as latency of completing such requests.

2.1.2 Transactional Coherence and Consistency

As its transactional programming model, Smart Memories implements transactional coherence and consistency (TCC). TCC leverages the same hardware components and operations used for cache coherence and streaming. The operations of the memory system in for implementing TCC protocol is described in this subsection.

As mentioned previously, TCC uses lazy conflict detection and eager commits. Transactional modifications are stored in processor's data cache and are broadcast to other transactions only after commit. Each read/write operation marks the accessed word in the data cache with an SR (speculatively read) or SM (speculatively modified) bits. Hence, transaction's read set and write set are completely specified within processor's data cache. Whenever the transaction finishes, it arbitrates for acquiring a commit token, which allows it to make its changes visible to others. If the transaction wins the arbitration, it asks protocol controller of its Quad to commit its write set.

Committing the write set of the transaction causes its modification to be written back to main memory as well as updating those words in other caches of the system. At the same time, the read set of other transactions are checked to detect any possible data dependence violation. A dependence violation is detected if the committing word is found in the data cache of another transaction and it is marked with the SR bit.

Similar to the coherence protocol, commit is also a two level process. Upon request from a committing transaction, a Quad's protocol controller uses its DMA engine to read the committing transaction's write set out of its data cache. For each word read, protocol controller then searches ("snoops") other data caches and updates the word if it is found. No action is taken if the word is not found in a data cache or if it is already is marked as SM (this means the other transaction has created its own copy of the data word). However, if the word is found and it is marked as SR, processor owning the cache is informed to abort the transaction and restart.

Next, words are sent to the owning memory controller and are written to main memory. The memory controller also broadcasts the words to other Quads within the system, so that they can also update their caches and check for violations. Figure 3 shows the steps involved in the process:

1- Word is read from committing processor's data cache
2- It is written to other caches in the Quad. SR bit of the word in those caches is checked.
3- Data is sent to owner memory controller
4- It is written to main memory
5- Memory controller broadcasts the word to all other Quads
6- Each Quad receives the word and writes it to its caches. SR bit of the word in each cache is checked to detect violations.

The commit procedure shares the same basic hardware mechanisms as cache coherence, such as cache searches (snoops) and broadcasting commit requests to all Quads. It also leverages the DMA engines, which are heavily used in the streaming mode, for reading data out of source cache. When implementing TCC, the protocol controller also satisfies data and instruction cache misses in basically the same way as for shared memory mode. The difference however is that the coherence protocol is turned off and the cache line is always directly fetched from off-chip memory.

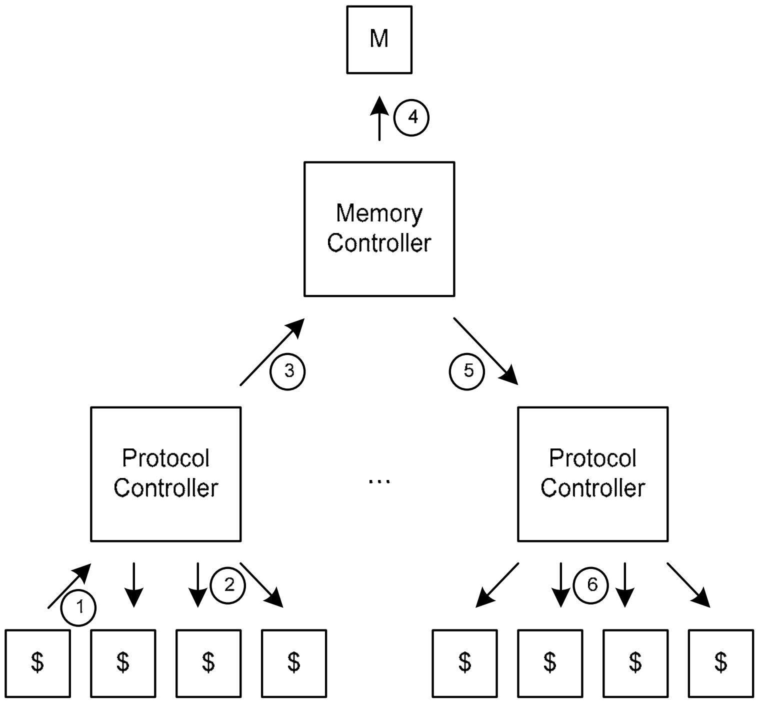


Figure 3 Steps for committing transactions write set

2.1.3 Fast, fine-grain synchronization protocol

To provide the programmer with the basic synchronization primitives such as locks and barriers necessary for parallel applications, Smart Memories implements a set of atomic read-modify-write operations in the memory system. These operations leverage the state bits associated with the data words, essentially treating them as lock bits. Necessary instructions are added to the processor's ISA to enable such operations.

When accessed by a synchronization operation, a word has two additional state bits: Full/Empty (F/E) and Wait (W) bits. The F/E bit essentially is an indication of whether the data word is currently full or empty. If the word is empty, its content cannot be read, and if it is full, it cannot be written. Hence, synchronization operations may succeed or fail based on the current value of this bit. The Wait bit indicates that there has been an unsuccessful attempt for reading or writing this word and that whenever a synchronization operation succeeds, it has to wake up the previous unsuccessful access so that it can retry reading/writing the word.

These bits are updated atomically by the synchronization access and processors also send out messages to the home memory controller of the word to report a successful or unsuccessful synchronization operation. There are two types of messages: a sync miss message indicates that a processor had an unsuccessful access and is stalled. A wakeup notification indicates that there was a successful synchronization access and a sleeping reader or writer needs to be awakened to retry its access. Table 1 summarizes synchronization operations defined for the processors, their semantics, and messages sent to the memory controller in each case. These instructions and state bits are discussed further in the later sections of this report.

Instruction	**Current**		**New**		**Operation**
	F/E	W	F/E	W	
Sync Load	0	X	0	1	Stall processor, Send sync load miss request
	1	0	0	0	Read data
	1	1	0	0	Read data, Send writer wakeup notification
Sync Store	0	0	1	0	Write data
	0	1	1	0	Write data, Send reader wakeup notification
	1	X	1	1	Stall processor, Send sync store miss request
Reset Load	X	0	0	0	Read data
	X	1	0	0	Read data, Send writer wakeup notification
Set Store	X	0	1	0	Write data
	X	1	1	0	Write data, Send reader wakeup notification
Future Load	0	X	0	1	Stall processor, Send future load miss request
	1	0	1	0	Read data
	1	1	1	1	

Table 1 Semantics of Synchronization Operations

Note that Reset Load and Set Store operations are always successful. They do not pay attention to the value of the F/E bit and always read/write the data word. Future load operation is the same as Sync Load, the only difference is that it does not reset the F/E bit after reading data word, hence does not “consume” the data and therefore it never sends a wakeup notification.

Sync miss messages include the address of the data word, the type of operation (sync load, sync store, future load) and the ID of the unsuccessful. Based on the address, these messages are routed to the appropriate memory controller and are queued until they are awakened.

Wakeup notifications carry the address of the word and the type (reader/writer). Depending on the type of the wakeup, the memory controller selects a sync miss from its queue that has the same address and sends it back to the protocol controller. The protocol controller retries the synchronization operation on behalf of the issuing processor and if it is successful, it notifies and un-stalls the processor. Figure 4 shows a simple example of how the protocol works:

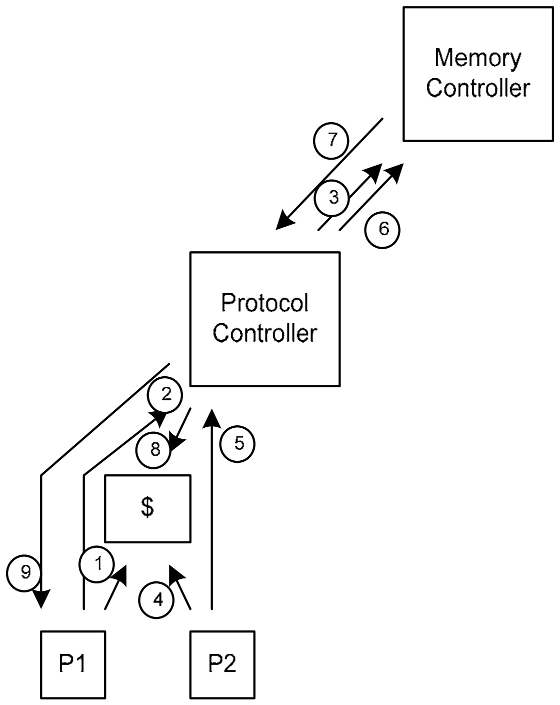


Figure 4 Steps in enforcing coherence

1- Processor P1 accesses a word with a sync load operation and stalls. It sets W=1.
2- It sends a sync load miss message to protocol controller.
3- Message is sent to memory controller and it is queued.
4- Processor P2 accesses the same word with a sync store operation, successfully writes data word, sets F/E=1 and observes W bit to be one.
5- It sends a reader wakeup notification to protocol controller
6- Wake up is sent to memory controller
7- Memory controller takes P1's sync load out of the queue and sends a replay request to protocol controller
8- Protocol controller replays sync load on behalf of P1, reads the data word and clears F/E bit.
9- It returns the data word back to P1 and un-stalls the processors

Synchronization accesses can be issued to both cached and un-cached addresses. When issued to a cached address, all synchronization operations are considered as writes, which need to acquire ownership of the cache line before they can attempt to read or write the word. The synchronization protocol relies on the underlying coherence mechanisms to refill the cache line with proper ownership before it attempts the synchronization access.

For un-cached addresses, synchronization accesses can only be issued to on-chip memories. They are simply routed to the Quad which contains the word and are replayed by its protocol controller on behalf of the issuing processors. Sync misses are still sent to memory controllers and queued there.

2.2 Processor

2.2.1 Overview of Tensilica LX

Tensilica provides the configurable embedded Xtensa processor. Tensilica's Xtensa Processor Generator automatically generates a synthesizable hardware description for the user customized processor configuration. The user can select pre-defined options such as floating-point co-processor (FPU) and can define custom instruction set extensions using the Tensilica Instruction Extension language (TIE).

The base Xtensa architecture is a 32-bit RISC instruction set architecture (ISA) with 24-bit instructions and windowed general-purpose register file. Register windows are 16-register wide. The total number of physical registers is 32 or 64.

The base Xtensa ISA pipeline is either five or seven pipeline stages and has a user selectable memory access latency of one or two. Two cycle memory latency allows designers to achieve faster clock cycles or to relax timing constraints on memories and wires.

The core supports some predefined options and ISA extensions:

- 16-bit wide instruction option for code density;
- 16-bit integer multiply-accumulator;
- 32-bit integer multiplier;
- 32-bit integer divider;
- 32-bit floating-point co-processor;
- 64-bit floating-point accelerator;
- 128-bit integer SIMD unit;
- configurable interrupts and timers;
- on-chip debug (OCD) port (via JTAG);
- instruction trace port.

Tensilica gives users a number of memory options:

- big or little endian;
- configurable width of load/store unit: 32/64/128;

- configurable instruction fetch width: 32/64;
- configurable instruction and data caches:
  - size (0-32KB) ;
  - associativity (1-4);
  - cache line size (16/32/64B);
  - write-through or write back;
- optional data RAM and/or ROM (0-256KB);
- optional instruction RAM and/or ROM (0-256KB);
- optional Xtensa Local Memory Interface (XLMI);
- optional Processor Interface (PIF) to external memory system;
- parity or Error Correction Code (ECC) options.
-

In addition to pre-defined options and extensions, a user can define custom processor extensions using the TIE language. TIE permits addition of registers, register files, and new instructions to improve performance of the most critical parts of the application. Multiple operation instruction formats can be defined using the Flexible Length Instruction eXtension (FLIX) feature to further improve performance.

Another feature of the TIE language is the ability to add user-defined processor interfaces such as simple input or output wires, queues with buffers, and lookup device ports. These interfaces can be used to interconnect multiple processors or to connect a processor to other hardware units.

The TIE compiler generates a customized processor, taking care of low-level implementation details such as pipeline interlocks, operand bypass logic, and instruction encoding.

Tensilica also provides customized software tools and libraries:

- instruction set simulator:
  - standalone single processor cycle-accurate simulator for performance modeling;
  - standalone single processor fast functional simulator for software development;
  - processor model for user-designed multi-processor simulator (through Tensilica's XTMP API);
- software development tools (based on GNU software tool chain):
  - optimizing C/C++ compiler;
  - linker;
  - assembler;
  - debugger;
  - profiler;
- XPES compiler: generator of application-specific TIE extensions;
- standard software libraries (GNU libc) for application development.

2.2.2 Interfacing Tensilica processor to Smart Memories

Connecting the Tensilica processor to the reconfigurable memory system is complicated because Tensilica interfaces were designed for different applications. Figure 5 shows all available memory and interface options. Although Xtensa processor has interfaces to implement instruction and data caches, these options do not support the functionality and flexibility necessary for Smart Memories architecture. For example, Xtensa caches do not support cache coherence. Cache interfaces are connected directly to SRAM arrays for cache tags and data, and the processor contains all the logic required for cache management. As a result, it is impossible to modify the functionality of the Xtensa caches or to re-use the same SRAM arrays for different memory structures like local scratchpads.

In addition to simple load and store instructions, the Smart Memories architecture supports several other memory operations such as synchronized loads and stores. These memory operations can easily be added to the instruction set of the processor using TIE language but it is impossible to extend Xtensa memory interfaces to support such instructions.

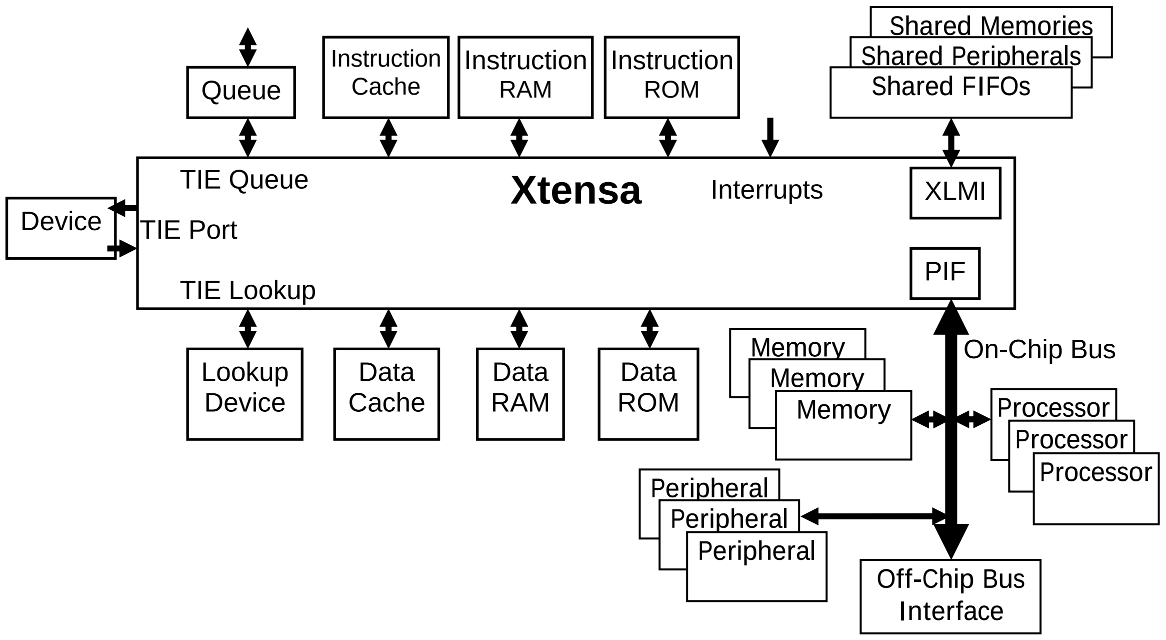


Figure 5 Xtensa Interfaces

Instead of cache interfaces we decided to use instruction and data RAM interfaces as shown in Figure 6. In this, case instruction fetches, loads and stores are sent to interface logic (Load Store Unit) that converts them into actual control signals for memory blocks used in the current configuration. Special memory operations are sent to the interface logic through TIE lookup port which has the same latency as the memory interfaces. If the data for a processor access is ready in 2 cycles, the interface logic sends it to the appropriate processor pins. If the reply data is not ready due to cache miss, arbitration conflict or remote memory access, the interface logic stalls processor clock until the data is ready.

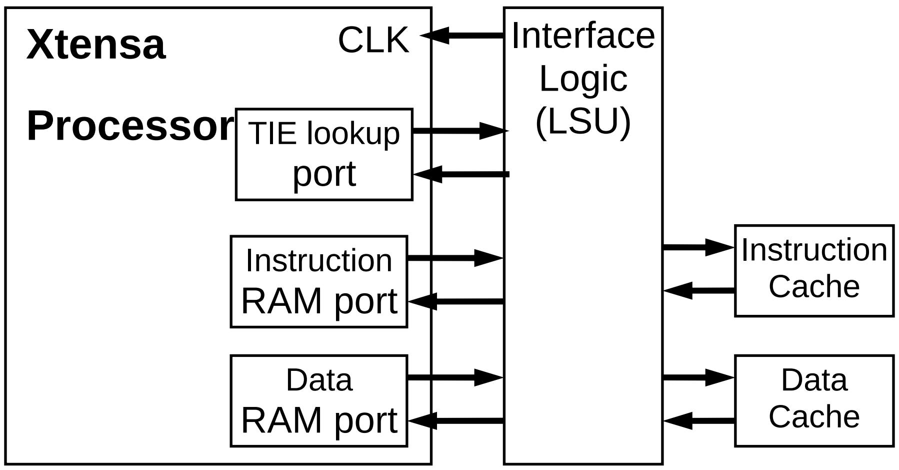


Figure 6 Processor Interfaces to Smart Memories

The advantage of this approach is that the instruction and data RAM interfaces are very simple: they consist of enable, write enable/byte enables, address and write data outputs and return data input. The meaning of the TIE port pins are defined by instruction semantics described in TIE. Processor logic on the critical path is minimal. Interface logic is free to perform any transformations with the virtual address supplied by the processor.

Special load instructions such as synchronized loads supported by Smart Memories are different from ordinary load instructions in that they can have side effects, i.e. alter architectural state of the memory. Standard load instructions do not have side effects, i.e. do not alter architectural state of the memory system, and therefore they can be executed by the processor as many times as necessary. This can happen because of processor exceptions as shown in Figure 7: loads are issued to the memory system at the end of E stage, load data is returned to the processor at the end of M2 stage, while the processor commit point is in W stage, i.e. all processor exceptions are resolved only in W stage. Stores are issued only in W stage after commit point.

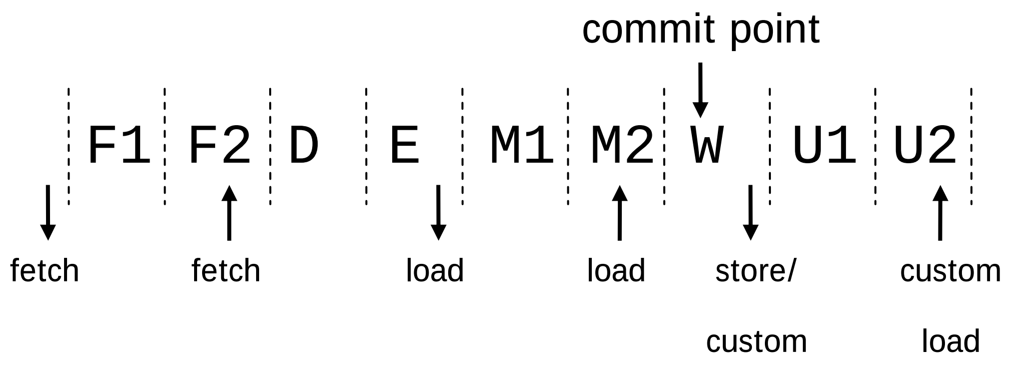


Figure 7 Processor Pipeline

Since it is very difficult to undo side effects of special memory operations, they are also issued after commit point in W stage. Processor pipeline was extended by 2 stages (U1 and U2 in Figure 7) to have the same 2 cycle latency for special load instructions.

However, having different issue stages for different memory operations creates the memory ordering problem as illustrated in Figure 8a. A load following synchronized load in the application code is seen by the memory system before the synchronized load because it is issued in the E stage. To prevent such re-ordering, we added pipeline interlocks between special memory operations and ordinary loads and stores. An example of such interlock is shown in Figure 8b. The load is stalled in the D stage for 4 cycles to make sure the memory system sees it 1 cycle after previous synchronized load. One extra empty cycle is added between 2 consecutive operations to simplify memory system logic for the case of synchronization stalls.

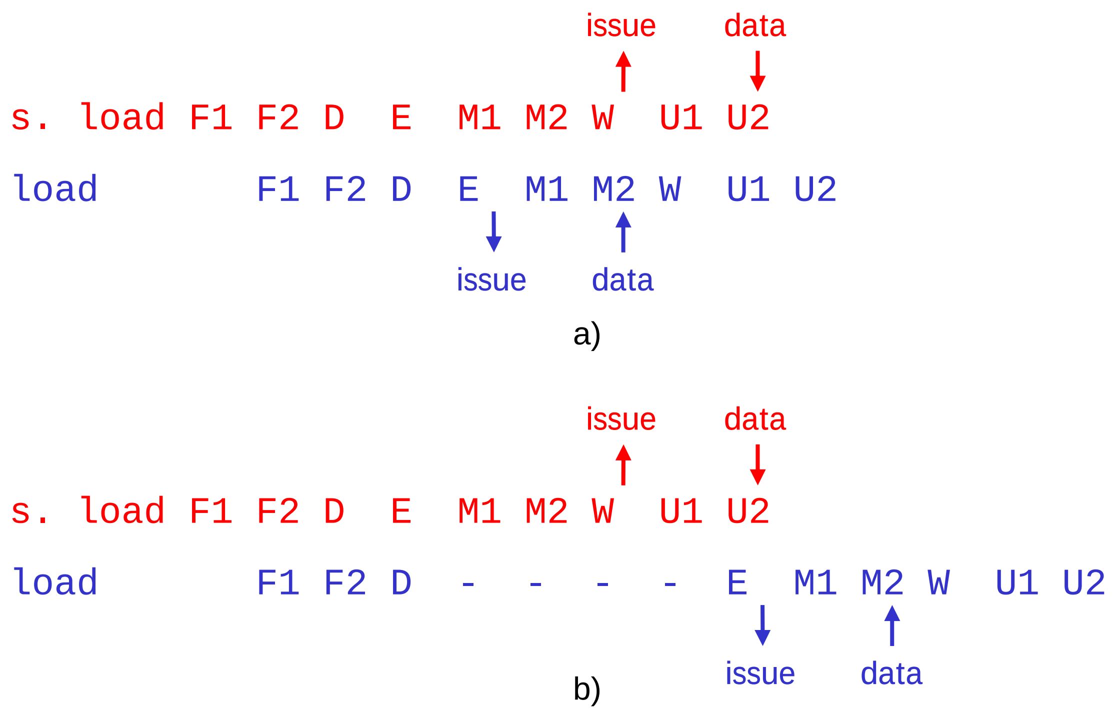

issue
data
s. load F1 F2 D E M1 M2 W U1 U2
load F1 F2 D E M1 M2 W U1 U2
issue
data
a)
issue
data
s. load F1 F2 D E M1 M2 W U1 U2
load F1 F2 D - - - - E M1 M2 W U1 U2
issue
data
b)

Figure 8 Memory operation pipeline: a) without interlocks; b) with interlocks

Another issue is related to very tight timing constraints on the processor clock signal as shown in Figure 9. The forward path for the memory operation data issued by the processor is going through the flop in the interface logic and then through the flop in the memory mat. In the reverse path the output of memory mat goes to the stall logic and determines whether the processor clock should be stalled or not. To avoid glitches on the processor clock the output of the stall logic must go through a flop clocked with inverted clock. The whole reverse path including memory mat, crossbar and stall logic delays should fit in a half clock cycle. This half cycle path is the most critical in the whole design and determines clock cycle time.

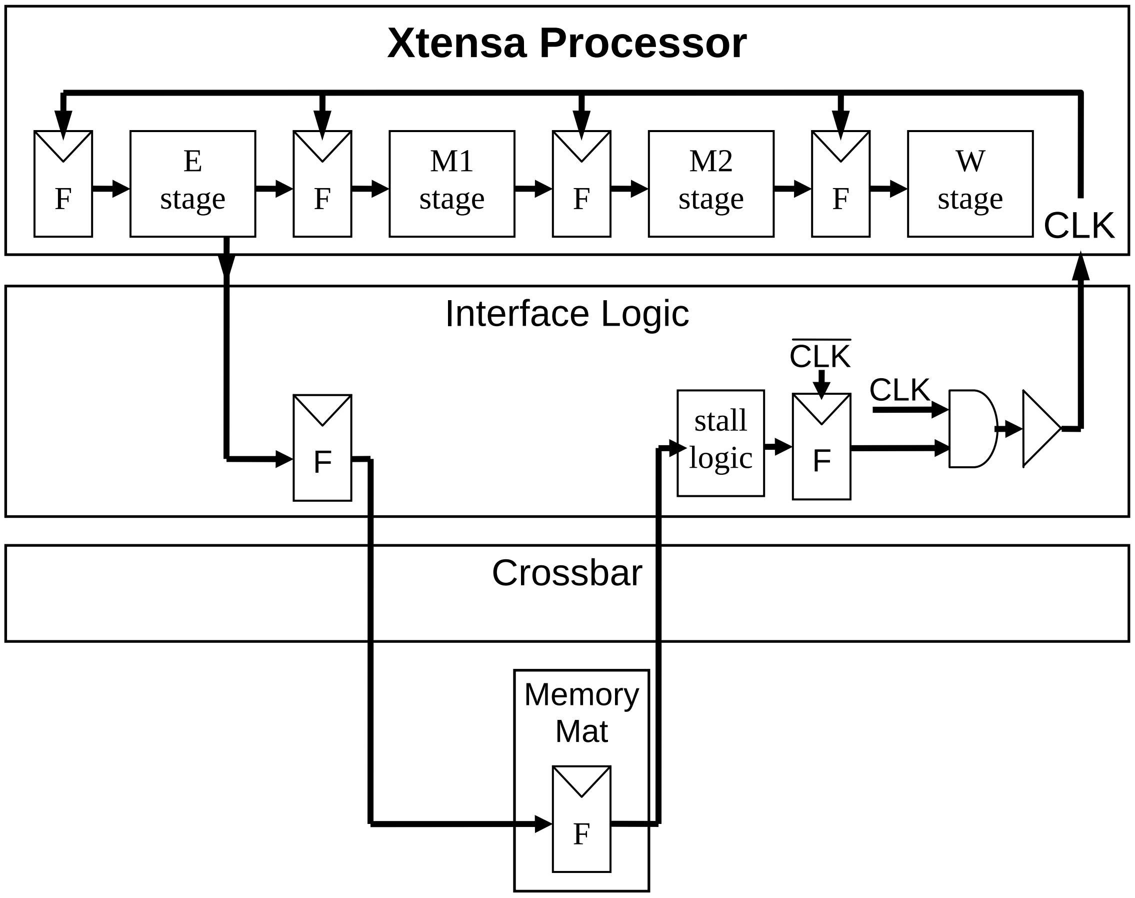


Figure 9 Processor and Tile Pipeline

To relax timing constraints, the processor is clocked with inverted clock: the reverse path delay becomes the whole clock cycle, rather than just the half cycle.

2.2.3 Special Memory Access Instructions

Several instructions were added to the Tensilica processor to exploit functionality of Smart Memories architecture:

synchronized load: stall if *full/empty* (FE) bit associated with data word is zero ("empty"), unstall when FE bit becomes one ("full), return data word to the processor and flip atomically FE bit to zero;

synchronized store: stall if FE bit is 1, unstall when it becomes 0, write data word and flip atomically FE bit to 1;

future load: the same as synchronized load but FE bit is not changed;

reset load: reset FE bit to 0 and return data word to the processor without stalls regardless of the state of FE bit;

set store: set FE bit to 1 and write data word without stalls;

meta load: read the value of *meta data* bits associated with data word;

meta store: write to meta data bits;

raw load: read data word skipping virtual-to-physical address translation, i.e. effective address calculated by the instruction is used as physical address directly;

raw store: write data word skipping virtual-to-physical address translation;

raw meta load: read meta data word skipping virtual-to-physical address translation;

raw meta store: write meta data word skipping virtual-to-physical address translation;

fifo load: read a value from a memory mat configured as a FIFO, FIFO status register in the interface logic is updated with FIFO status information, i.e. whether FIFO was empty;

fifo store: store a value to a FIFO, FIFO status register is updated with FIFO status information, i.e. whether FIFO was full;

safe load: read a data word from the memory and ignore virtual-to-physical address translation errors;

memory barrier: stall the processor while there are outstanding memory operations, i.e. non-blocking stores;

hard interrupt acknowledgement: signal to the memory system that hard interrupt was received by the processor, this instruction is supposed to be used only inside interrupt handler code;

mat gang write: gang write all meta data bits in the memory mat, supported only for 3 meta data bits;

conditional mat gang write: conditionally gang write all meta data bits in the memory mat, supported only for one meta data bit;

cache gang write: gang write all meta data bits in the data cache, supported only for 3 meta data bits;

conditional cache gang write: conditionally gang write all meta data bits in the data cache, supported only for one meta data bit.

These instructions use TIE lookup port to pass information from processor to the memory system as described in the previous section.

2.2.4 Pre-Defined and VLIW Processor Extensions

To increase the computational capabilities and usability of the Smart Memories architecture, the following pre-defined processor options were selected:

- 32-bit integer multiplier;
- 32-bit integer divider;
- 32-bit floating point unit;
- 64-bit floating point accelerator;
- 4 scratch registers;
- On-Chip Debug (OCD) via JTAG interface;
- instruction trace port;
- variable 16/24/64-bit instruction formats for code density and FLIX/VLIW extension.

To further improve performance of the processor and utilization of the memory system, we added several multi-instruction formats using FLIX/VLIW capability of Tensilica system:

- {ANY; INT; FP};
- {ANY; NOP; FP};
- {ANY; INT; LIMITED INT};

where ANY means any type instruction, INT means integer instruction type (excluding memory operations), FP means floating-point instruction type, LIMITED INT means a small subset of integer instructions which require at most 1 read and 1 write port.

The reason for this choice of instruction formats is the limitation of Xtensa processor generator: register file ports can not be shared between different slots of FLIX/VLIW format. For example, FP multiply-add instruction requires 3 read and 1 write ports, if such operation can be present in 2 different slots, then FP register file must have at least 6 read and 2 write ports even if 2 such operations are never put in the same instruction. On the other hand, memory operations can only be allocated in slot 0 and the common usage case is to have memory operation and compute operation such as multiply-add in the same instruction. This means that it should be possible to have FP operations in slots other than 0 but the number of such slots should be minimal.

2.2.5 Processor Extension for Recovery from Missed Speculation

To be able to restart execution of a speculative transaction after violation, the system state must be saved at the beginning of the transaction. There are two distinct components of system state in the Smart Memories architecture: memory system state and the processor state. Memory system state can be quickly restored to the check point because all speculative changes are buffered in the data cache and can be easily erased by invalidating cache lines with gang write operations.

Processor state consists of the general purpose register files (integer and floating point) and various control registers. Our approach is to force the compiler to spill general purpose registers into the stack memory at the transaction check point using the asm volatile construct. After a check point, the compiler inserts load instructions to reload the values into the register files. The advantage of this approach is that compiler spills only live register values, minimizing the number of extra load and store instructions.

Spilled register values in the memory are check-pointed using the same mechanism as other memory state. The only general purpose register that can not be check-pointed this way is the stack pointer register a1; we use a separate mechanism for the stack pointer as well as other processor control registers.

To save the state of control registers we added 3 more registers and used one of the scratch registers:

- SPEC_PS – a copy of PS (processor status) register;
- SPEC_RESTART_ADDR – transaction restart address;
- SPEC_TERMINATE_ADDR – address to jump to in case of execution abort;
- MISC1 – stack pointer.

To use these registers in interrupt handlers we added 2 special return-from-interrupt instructions:

- SPEC_RFI_RESTART – return from interrupt to the address stored in SPEC_RESTART_ADDR register, SPEC_PS register is copied atomically to PS;
- SPEC_RFI_TERMINATE – the same except that SPEC_TERMINATE_ADDR register is used as return address.

2.3 Memory Mat and Crossbar

A memory mat is the basic unit of storage in the Smart Memories system. In addition to storing bits of information, it is also capable of performing very simple bit manipulation operations on some of the stored bits. Depending on the configuration, a memory mat can be used as simple local storage, a hardware FIFO, or as part of a cache for storing either tag or data. Each Tile has 16 memory mats which are connected to processors and outside world by a crossbar interconnect.

This section describes the internal architecture and operations of the memory mat. It also describes how memory mats are aggregated and used to implement more sophisticated storage structures, such as normal or transactional caches. The operations of the crossbar and inter-mat communication network (IMCN), which exchanges control information between different mats when implementing composite storage structures are also explained in this section.

2.3.1 Memory mat organization

Figure 10 shows the internal architecture of memory mat at a high level. Major blocks and the flow of data and address information are shown. Each block is capable of performing a certain set of operations, which are described in the more detail in the following sections.

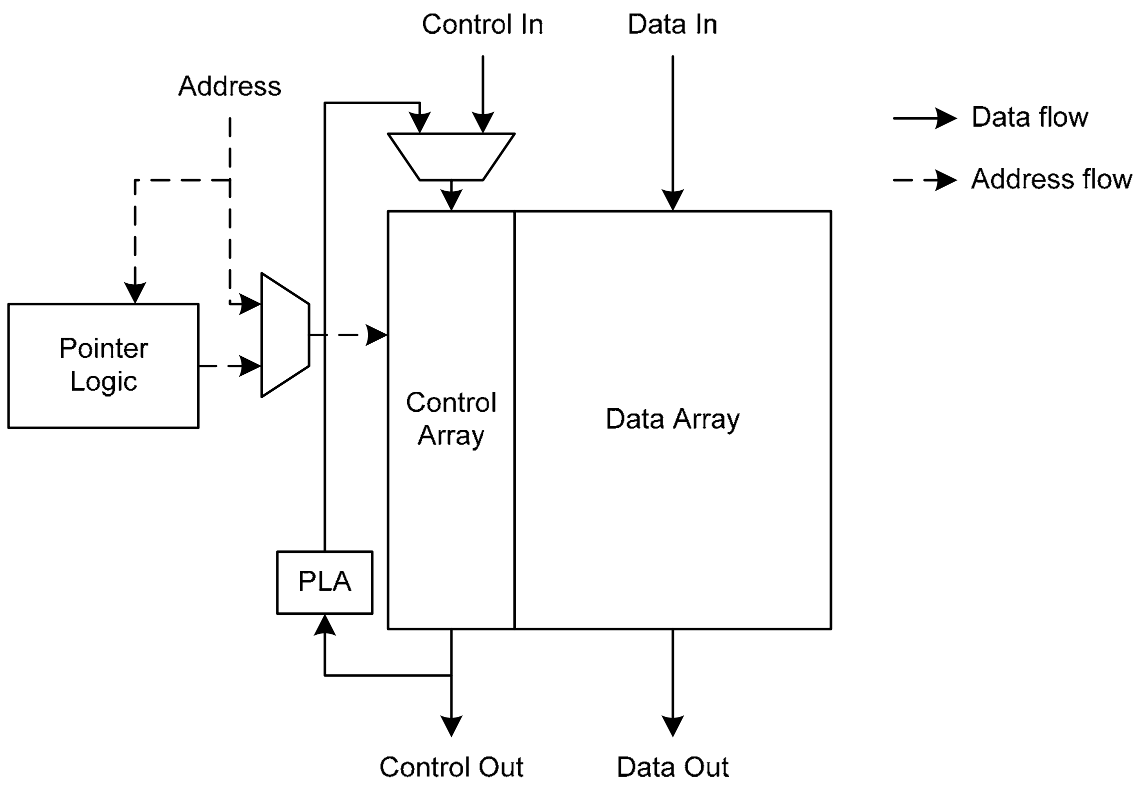


Figure 10 Major Blocks of the Memory Mat

2.3.2 Data Array

Data array (or data core) is shown in Figure 11. It has 1024 entries of 32-bit words and is capable of doing read, write and compare operations on the accessed word. There is a 4-bit mask input into the array that allows each byte within the 32-word to be written independently. In order to do comparison operations, the array is equipped with a 32-bit comparator, which compares contents of the word with the information provided on the Data In input and gives out a Data Match signal. This data match signal is sent out to the processors over the crossbar as well as passed to control array logic.

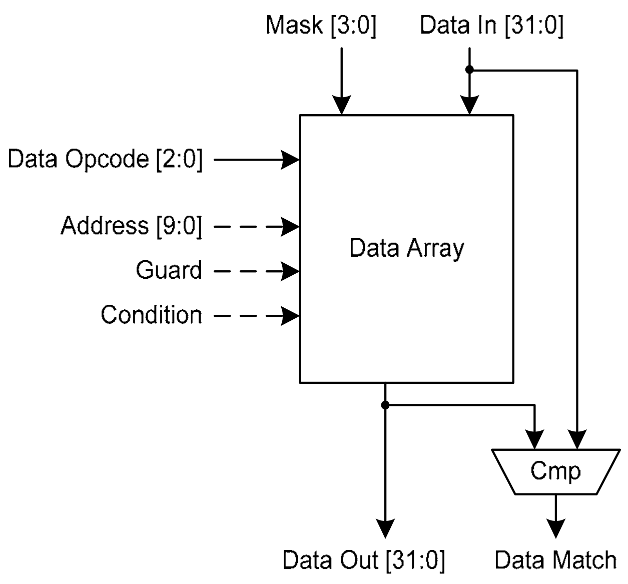


Figure 11 Data array. Solid lines are mat input/output signals while dashed lines are internal control signals

In addition to the Address and Data In, data array receives a 3-bit Data Opcode which dictates what operation should be performed on the addressed word. Furthermore, write operations in the data array can be “Guarded” or “Conditional”. Such operations are particularly useful when implementing caches for example: the data storage can discard cache write operation if tag storage repots a cache miss after tag comparison. For performing such operations, data array receives two additional control bits, Guard and Condition, and can decide to discard a write operation if either of the Guard or Condition signals is not active. How the Guard and Condition signals are generated is discussed later in this section. Table 2 lists all operations of the data array.

Data Opcode	Operation
3’h0 – Nop	Array is idle, not doing anything
3’h1 – Unused	Equivalent to Nop
3’h2 – Read	Addressed word is read
3’h3 – Compare	Addressed word is read and compared with Data In
3’h4 – (Guarded) Write	Addressed word is written, only if Guard signal is active

3'h5 – (Guarded) Conditional Write	Address word is written, only if Guard and Condition signal are both active
3'h6 – Unguarded Write	Addressed word is written
3'h7 – Unguarded Conditional Write	Addressed word is written if Condition signal is active

Table 2 Operations in data array

2.3.3 Control Array

Control array (or control core) is a 1024 entry 6-bit array, where every entry corresponds to an entry in the data array (Figure 12). These 6-bits are called meta-data bits or control bits associated with each 32-bit word. Control array is dual-ported: it can do a read and a write operation in the same cycle. This allows the control array to do atomic read-modify-write operation on the control bits. The read address is always generated from the main address input of the mat. The write address can either be derived from main address input or internally generated when doing read-modify-write operations. An internal forwarding logic bypasses write values to read port if subsequent read operation goes to the address that was written previous cycle.

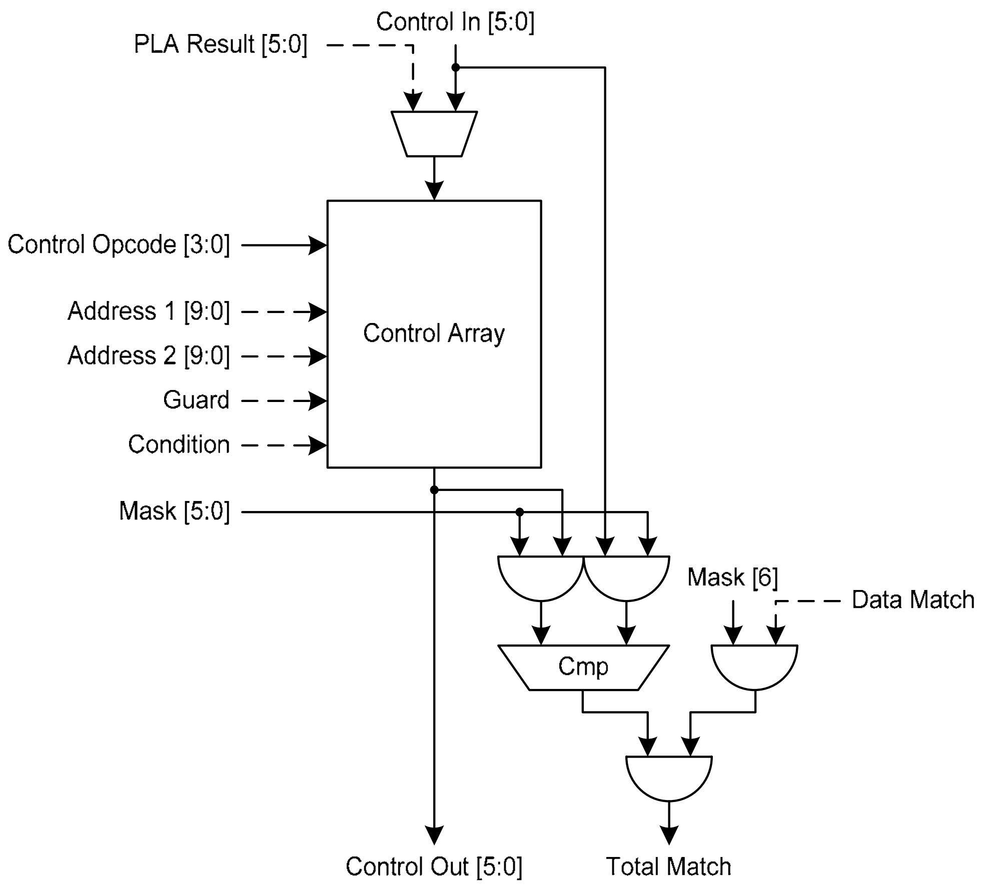


Figure 12 Control array. Solid lines are mat input/output signals while dashed lines are internal signals generated inside mat

Control array is accessed along with the data array and is capable of doing the same operations as data array: read, write and compare. Furthermore, it can do read-modify-write and compare-modify-write operations by writing back new values for the control bits supplied by an internal PLA logic. It receives the same Guard and Condition signals as data array and has different flavors of write operations. For compare operations, control array compares the contents of the accessed entry with the Control In input. A Total Match signal is generated as result. There is a 7-bit mask which indicates which control bits participate in the compare operation and which control bits are ignored. MSB of the mask signal indicates whether the result of the comparison in data array (Data Match) should participate in generating the Total Match signal.

Three bits of the control array (bits 0-2) are accessible by column-wise gang operations: a whole column can be set to one or zero in a single-cycle operation. Also, one column of the array (bit 2) is capable of doing conditional gang write operation: Bit 2 in each entry of the array can be written with one or zero, given that bit 1 of the same entry is set to one. These operations are mainly used when implementing transactional caches: all transactional modifications can be flushed away in a single cycle.

Control array operations are dictated by a 4-bit Control Opcode input. Table 3 lists all the operations on the control array.

Control Opcode	Operation	Example Usage
4’h0 – Nop	Array is idle, not doing anything	
4’h1 – Unused	Same as Nop	
4’h2 – Unguarded Read-Modify-Write	Control bits are read and new values are written back the next cycle	Cache line eviction
4’h3 – (Guarded) Compare-Modify-Write	Control bits are read and compared with Control In. New values are written back only if Guard signal is active.	Change cache line state to Modified in case of cache writes
4’h4 – Read	Control bits are read and sent out on Control Out	Reading cache line state
4’h5 – Compare	Control bits are read and compared with Control In	Checking cache line state (Valid, etc.)
4’h6 – (Guarded) Read-Modify-Write	Control bits are read and supplied on Control Out. New values are written back only if Guard signal is active	Coherence operations (invalidate, degrade)
4’h7 – (Guarded) Conditional Read-Modify-Write	Control bits are read and sent out. New values are written if both Guard and Condition signals are active	
4’h8 – (Guarded) Write	If Guard signal is active, control bits are written with the Control In value	
4’h9 – (Guarded) Conditional Write	If both Guard and Condition signals are active, control bits are written with the Control In value	
4’hA – Unguarded Write	Control bits are written by the value of Control In	Updating cache line state

4’hB – Unused	Same as Nop	
4’hC – Gang Write	If Address 0, 1 or 2 is one, column 0, 1 or 2 of control array receives value of Control In 0, 1 or 2.	Cache flush
4’hD – Conditional Gang Write	If Address 2 is one, bit 2 of each entry receives value of Control In 2 Only if bit 1 in the same entry is one	Flush of speculatively modified lines
4’hE – Conditional Unguarded Write	Writes bits in the control array only if Condition signal is active	
4’hF – Unused	Same as Nop	

Table 3 Operations in control array

2.3.4 PLA block

A small PLA block allows mat to perform read-modify-write operations on the control bits associated with every data word. The PLA operates on the control bits read from control array and generates new values, which are written back to control array the next cycle. Forwarding logic takes care of the case when the same entry in the array is immediately read after being written, hence the read-modify-write operations takes place atomically from the user’s point of view.

PLA logic is controlled by a 4 bit PLA Opcode signal which is supplied to the mat. This opcode dictates the logic function that needs to be performed on the control bits. In general, PLA logic receives the following inputs to operate upon:

Control [5:0]: six control bits from control array

Data match: result of the comparison operation in the data array

Total match: logical AND of Data match with Control match, which is result of comparison operation in the control array

IMCN [1:0]: Inter-Mat Communication Network mat inputs (explained later)

PLA Opcode [3:0]: mat inputs

Byte Write: generated based on the write byte mask in the data array, indicates whether the write operation writes the whole word or not

2.3.5 Pointer logic

Each memory mat is equipped with a pair of pointers which allows it to be used as a hardware FIFO (Figure 13). An external output, FIFO select, dictates whether mat should use the externally supplied “Address In” signal or use internal pointers to generated address for data and control arrays. These pointers are automatically incremented after each access: read and compare operations increment head pointer, while write operations increment tail pointer. Increment of the tail pointer can be guarded the same way that a write operation is guarded: if the guard signal is not active, the pointer will not be incremented. An example usage of guarded increment is described later in this section, when explaining how a transactional cache is configured.

The depth of the FIFO also can be controlled via a configuration register. Whenever size of FIFO grows to the value of the depth register and user tries to write the FIFO, write operation is ignored and a FIFO Error output signal is asserted. The same situation happens if user tries to read an empty FIFO. Also, there is a threshold register which its value can be set by user. When the size of the FIFO grows to this threshold, a separate FIFO Full output signal is asserted to let the user know that FIFO is almost full.

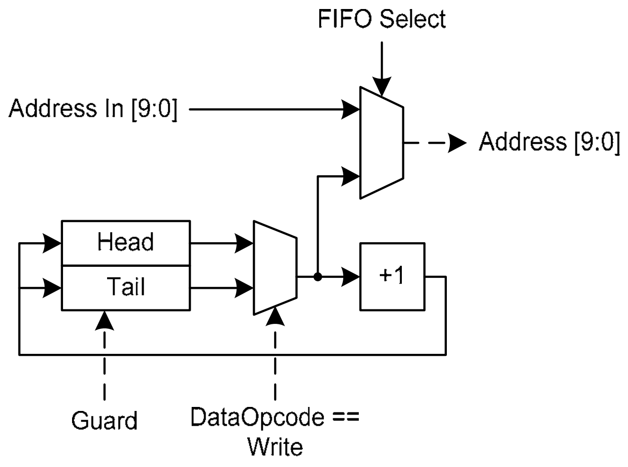


Figure 13 Pointer logic. Solid lines are main inputs/outputs of the mat, while dashed lines are internal signals

2.3.6 Guard and Condition Logic

As mentioned before, write operations in the data array and control array can be controlled by any combination of a Guard and Condition signals. This allows a mat to ignore write operations if necessary. The best example is when a combination of memory mats is used to implement a cache: When a write operation is issued to the cache, it should be ignored if cache line is not present in the cache or if it is in the correct state (e.g. it is in Invalid or Shared state). In this case, the hit/miss indication acts as the Guard signal, which orders the mat to discard the write operation. Guard signal can be configured to be any function of the IMCN_in inputs, while Condition can be any of the 6 control bits within the control array. An example usage of the Condition bit is when performing a synchronized store operation, which sets a Full/Empty bit after writing the data word. In this case, one of the control bit is used to implement the Full/Empty bit; if the Full/Empty bit is already set (data word is full), the write operation should be discarded.

2.3.7 Crossbar and Inter-Mat Communication Network (IMCN)

There are 16 memory mats within each Tile which are accessible by the two processors and from the outside of the Tile. Tile crossbar is the entity that connects the two processors and outside interface to the memory mats. In addition, memory mats have their own inter-mat communication network to exchange control information when implementing composite storage structures like caches.

Crossbar

The Tile crossbar (Figure 14) connects processors' load/store unit and protocol controller to memory mats. It performs the arbitration between different sources when accessing mats and issued grants. It is also capable of broadcasting operations to multiple mats, where an access is simultaneously sent to more than one memory mat, if necessary.

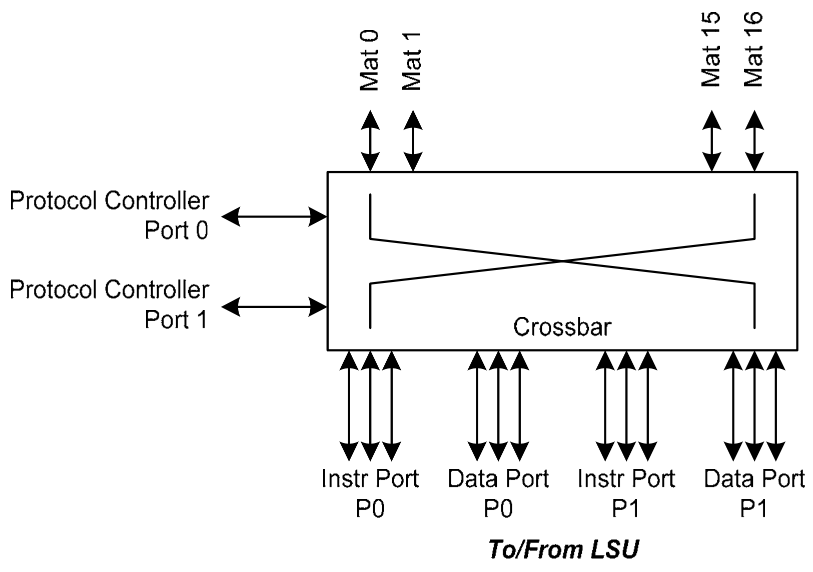


Figure 14 Tile crossbar

Each part of the LSU (processor 0 and processor 1) has two independent ports to the crossbar, one used for data accesses and the other for instruction accesses. When accessing a mat LSU provides the mat with all necessary signals: Address, Data In, Control In, Mask, FIFO select, Data Opcode, Control Opcode and PLA Opcode. We call the combination of these signals a “mat access”.

Each LSU port to the crossbar can access up to three independent set of mats: tag mat(s), data mat(s) and FIFO mat. Any combination of these mat accesses might be generated as result of the processor's instruction or data access. The data mat(s) contain actual data; they might be local scratchpads or mats that store the data portion of cache line in a cache configuration. The tag mat(s) contain the tag portion of the cache lines when mats are used to implement caches. The FIFO access is used in TCC mode only and is explained later. Each one of the data, tag and FIFO accesses have a 16-bit mat mask, which tells the crossbar which mats the access should be sent to.

The crossbar also acts as an arbiter, which resolves conflicts between the accesses from the two processors and protocol controller. The protocol controller is given priority at all times and its accesses always go through without being stalled. Since protocol controller might access memory mats in all of the four Tiles simultaneously, having higher priority enables it to statically schedule its accesses to memory mats and simplifies its internal pipelines. A fair round-robin arbitration mechanism resolves conflicts between the two processors and stalls the losing processor.

As shown in Figure 14, protocol controller has two independent ports for accessing the memory mats within the Tile. Like processor ports, each of these ports can broadcast an access to any number of memory mats independently. Note that crossbar assumes that these two ports never try to access the same memory mats at the same time, therefore there is no arbitration mechanism implemented between these two ports.

Inter-Mat Communication Network (IMCN)

Memory mats can be used to implement composite storage elements such as caches with different parameters such as size, ways and line size. In such configurations, some of the mats are used to store tag portion of the cache line as well as cache line state information, while other mats are used to store the data. In order for such configuration to work correctly, some control information needs to be exchanged between tag mats and data mats. For example, the hit/miss information should to be transmitted from tag mats to data mats, such that the data mats can ignore the operation if there was not a hit in that particular cache way. IMCN is a small network that allows mats to exchange such information efficiently.

Each mat supplies two bits to inter-mat communication network (IMCN_out [1:0]) and receives two bits from it (IMCN_in [1:0]). A configuration register within the mat dictates what values mat sends out on each of the IMCN outputs independently, which can be any of the following signals or their logical inverse:

Any of the control bits

Data Match (from data array)

Total Match

A set of configuration registers within the IMCN decides what values are supplied to memory mats on their IMCN_in inputs. Figure 15 shows the IMCN logic for generating these inputs: For each bit of the IMCN_in per memory mat, a mask register decides which mats participate in generating that input. Then a logical OR operation is performed on all the participating signals and result is sent to the receiving memory mat. This way, each mat can receive control information from any other mat or combination of mats on any of its IMCN_in inputs.

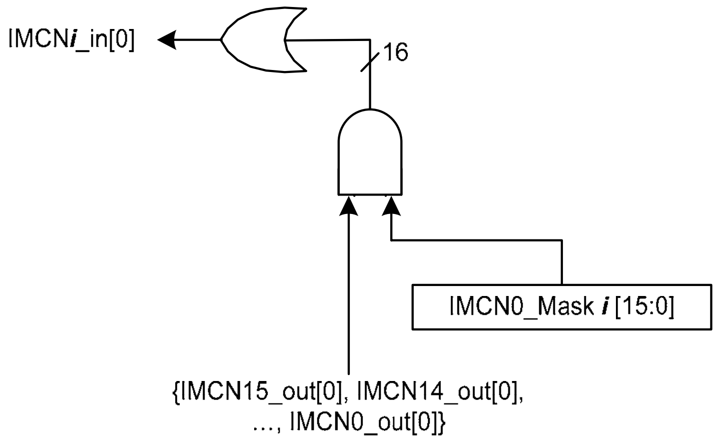


Figure 15 IMCN_in signal generation for each memory mat

2.3.8 Examples

In this subsection, configuration and operation of memory mats are described by two examples. The first example shows how to use a set of memory mats to implement a 16KB, 2-way set associative cache, while second example shows the configuration and operation of a transactional cache which is capable of tracking and storing speculative modifications.

A 2-way set associative cache

Figure 16 shows how a 2-way set associative cache is implemented using six memory mats. Each way of the cache consists of two mats storing the data part of the cache line and one mat storing the tag portion as well as cache line state information. Hit/miss information is sent across the IMCN from each tag mat to the corresponding data mats. Whenever LSU issues a cache operation, crossbar routes the tag access to both tag mats and data access to appropriate data mats (depending on the offset within the cache line, data access might go to Data 0 or Data 1).

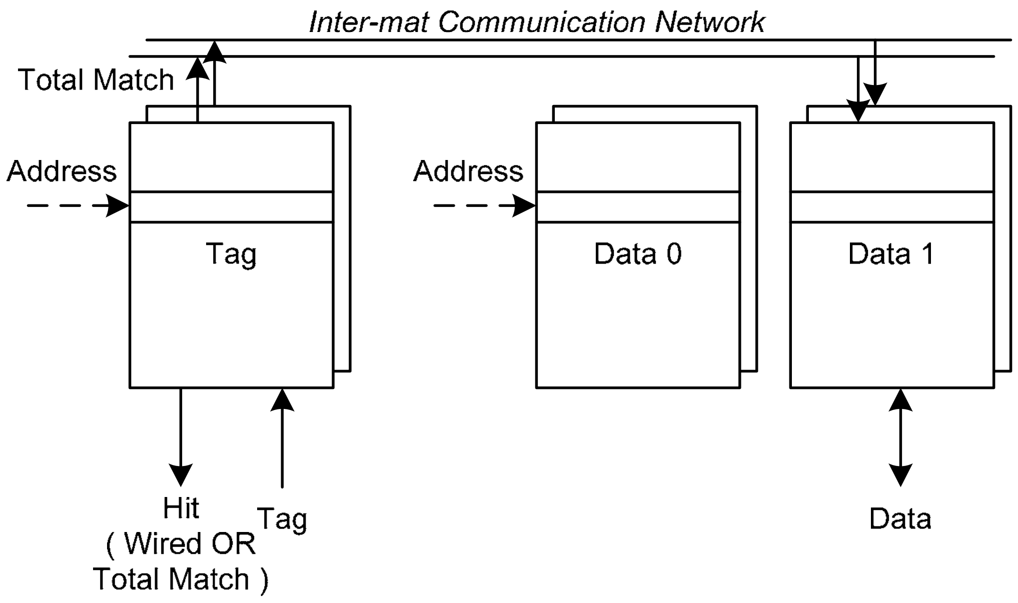


Figure 16 A sample 2-way set associative cache configuration

Meta-data bits in the tag mats are used to encode the state of the cache line according to the coherence protocol as well as information necessary for implementing desired line replacement policy. Assuming a simple MESI coherence protocol and Not-Most-Recently-Used replacement policy, the encoding cache line state is as follows:

Bit 0: unused

Bit 1: Shared/Exclusive

Bit 2: Valid/Invalid

Bit3: Modified

Bit4: MRU (Most Recently Used)

Bit 5: Reserved (Indicates a pending refill on this cache line)

All tag and data mats in the cache are accessed simultaneously by LSU. Tag mats perform a compare operation on both data and control arrays: Data In signal brings in the tag portion of the cache line address, while Control In brings the desired line state for comparison. The Mask signal is used to discard unwanted bits of the control array, for example when doing a cache read, value of the S/E or MRU bits is not important. The result of the comparison is reported by each way on its Total Match output and is returned to the LSU by crossbar. The Total Match output of each tag mat is also transmitted to its corresponding data mats over the IMCN so that data mats could ignore the operation if their associated tag mat did not have a hit. PLA operation within the tag mat updates the control bits accordingly: for example, in case of a cache write, Modified bit is turned on whenever there is a cache hit.

Data array in data mats receives a read or guarded write operation (depending on type of cache access, Load or Store) from crossbar. In case of read, each data mat returns the contents of the addressed location back to crossbar and crossbar selects the appropriate word depending on the hit signal from each tag mat. In case of write operations, the data array receives a guarded write operation and the Guard signal is configured to be the hit/miss signal received on the IMCN from the associated tag mat, hence if the corresponding tag mat does not report a hit the write operation is discarded and contents remain untouched.

The exact configuration of the cache (such as which mats are used for tag storage and which mats are used for data storage, number of ways in the cache, etc.) as well as the details of tag and data accesses (such as Data Opcode, Control Opcode, PLA Opcode, Mask, etc.) are stored in the configuration registers inside the Load/Store Unit and are discussed in detail in the next section.

Transactional cache

A transactional cache in the Smart Memories stores the speculative writes issued by the transaction. Data of the speculative write is stored in the data mats of the cache while a separate mat is used to store the address of all speculative writes. This list of addresses is then used by the protocol controller to commit all such modifications to main memory after transaction finishes. Figure 17 shows an example of a two way transactional cache.

The mapping of control bits for representing cache line state and line replacement information is as follows:

Bit 0: SR (Speculatively Read – indicates that the line has been read by the transaction)

Bit 1: SM (Speculatively Modified – indicates that the line has been speculatively modified by the transaction)

Bit 2: Valid/Invalid

Bit 3: Modified

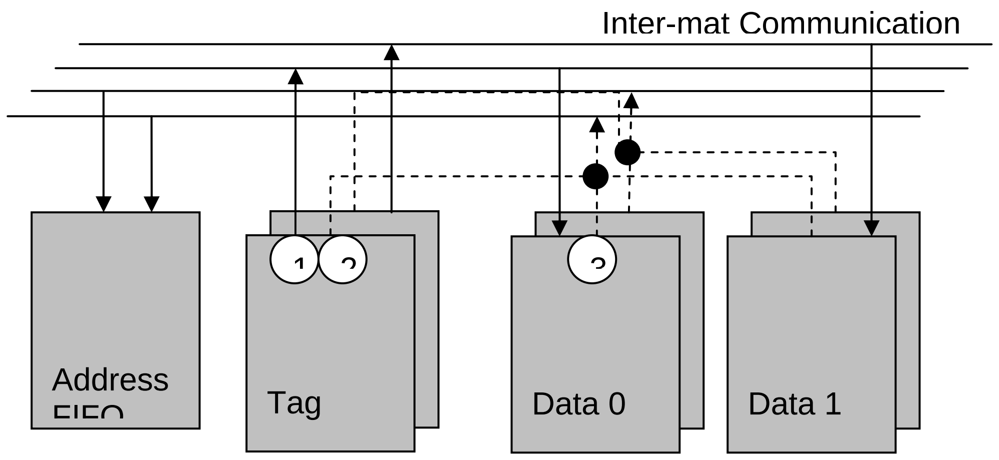
Bit 4: MRU (Most Recently Used)

Bit 5: Reserved

In addition, the control bits in the data mats are also used to mark each word independently as speculatively read or written:

Bit 0: SR (Speculatively Read)

Bit 1: SM (Speculatively Modified)



1) Total Match sent out by tag mats to the corresponding data mats
2) ~Total Match sent out to be ORed with SM bits from the data mat of the same way and then sent to FIFO
3) SM bits sent out from data mats to be ORed with ~Total Match from the same way and sent to FIFO

Figure 17 A transactional cache

The basics of the operations for the transactional cache are essentially the same as normal cache; the only difference is in that an additional memory mat is used in FIFO mode to store all the addresses written by the transaction. Whenever LSU issues a cache write operation, it also sends a FIFO access to the Address FIFO along with the tag and data accesses. Address FIFO receives the address of the word on its Data In input and writes it into the data array. The IMCN and Guard logic are configured such that the FIFO accepts and writes the address only if there is a cache hit and the word is not already marked as speculatively modified. This way, a word might be written multiple times during the transaction execution, but its address is placed only once in the Address FIFO.

In order to implement the guard logic, each tag mat sends the inverse (Not) of its Total Match signal on the IMCN_out, while each data mat sends out its SM bit. A logical OR operation is performed in the IMCN and FIFO mat receives (~Total Match | SM) from each way of the cache on one of its IMCN_in inputs. The Guard signal inside the FIFO mat is then configured according to the following table:

Guard	**IMCN_in[1]**	**IMCN_in[0]**
1	0	0

1	0	1
1	1	0

Table 4 Guard function for the Address FIFO

After the transaction ends successfully, protocol controller uses the contents of the Address FIFO to send all transaction's modifications to main memory and hence commit its state. SR and SM bits in both tag and data mats are flash-cleared using column-wise gang operations and a new transaction can start.

If a transaction needs to be restarted, all its speculative modifications have to be discarded. This is simply performed using conditional gang write operation in the control array: Valid bit in the tag mats is conditionally flash-cleared if SM bit is set, which invalidates all speculative modifications inside the cache. After that, SM and SR bits in both data and tag mats are flash-cleared and cache will be ready for transaction to restart.

2.4 Load/Store Unit (LSU)

The Load/Store Unit (LSU) connects processor cores to the Tile crossbar and provides flexibility in performing memory operations. It also has an interface to communicate with the protocol controller sitting outside the Tile. The LSU keeps details of the Tile's memory configuration as well as the exact behavior of each memory operation issued by the processor in a number of configuration registers. This section describes the major functions of the LSU, provides an overview of its configuration capabilities, and gives details of its communication with the protocol controller.

2.4.1 Interfaces

Figure 18 shows the interfaces of the LSU. On the bottom there are processor ports: instruction, data and TIE. The TIE port is used for issuing special memory operations which are added to processor's instruction set using TIE language. Data and TIE ports of the processor are 32 bits wide, while the instruction port is 64 bits wide. Each processor has its own separate ports to the LSU. On the top side, LSU connects to the crossbar ports and on the left to the protocol controller. Note that the two processors share the single interface to the protocol controller.

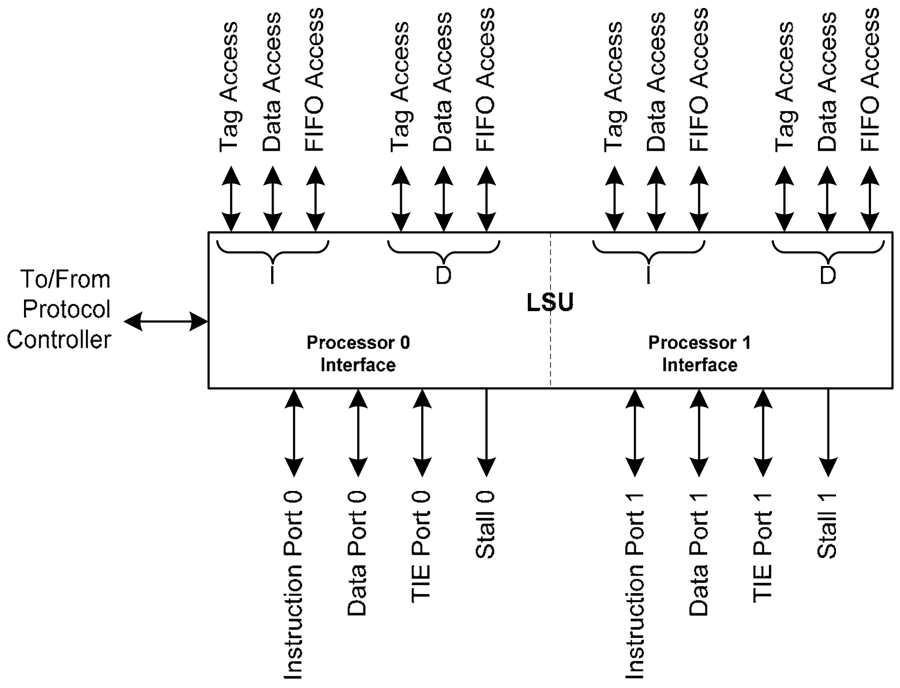


Figure 18 Interfaces to the Load/Store Unit

Processors are configured such that they always activate only one of their data or TIE ports in a cycle. However, accesses from the instruction port can occur simultaneously with a data or TIE port access. As indicated in the figure, the LSU also provides a mechanism to stall processors upon events such as cache misses or direct accesses to non-local memories (e.g. in other Tiles or Quads), which have to go through the protocol controller.

2.4.2 Tile memory configuration

As mentioned in the previous section, memory mats in the Tile can be configured differently depending on the memory model. There are two structures in the LSU that keep track of this information: the cache configuration registers and the segment table. The two processors in the Tile each have their own set.

A processor's instruction and data caches are specified by a collection of five registers inside the LSU: one register describes the main parameters of the cache, while four registers store information of up to four cache ways. The details of these registers are described below:

i/d_cache_way0-3_info

Bit [0]: way enable

Bit [4:1]: tag mat number

Bit [8:5]: starting data mat number

i/d_cache_info
Bit [1:0]: unused
Bit [3:2]: number of data mats in each way
Bit [5:4]: cache line size
Bit [7:6]: number of rows per cache line
Bit [11:8]: FIFO mat ID (for TCC caches only)

Each way of the cache, as described by the examples in the previous section, consists of a tag mat and a number of data mats. Each way info register determines whether a specific way is enabled in the cache and which mats serve as its tag and data storage. When a cache way has more than one data mat, only the ID of the first data mat is stored in the register; other data mats simply follow the starting mat sequentially. The number of data mats in each cache way is stored in bits 3-2 of the cache info registers.

Size of the cache line can be 16, 32, 64 or 128 bytes and is stored in bits 5-4 of the cache info register. Larger cache lines span multiple data mat rows, so the number of rows per cache line is specified separately. For example, a cache with two data mats in each way and a line size of 16 bytes will use two rows of the mats per line, as shown in Figure 19. Thus, each index of the tag mat is associated with two different indices of the data mats and hence the LSU calculates tag mat index and data mat index separately.

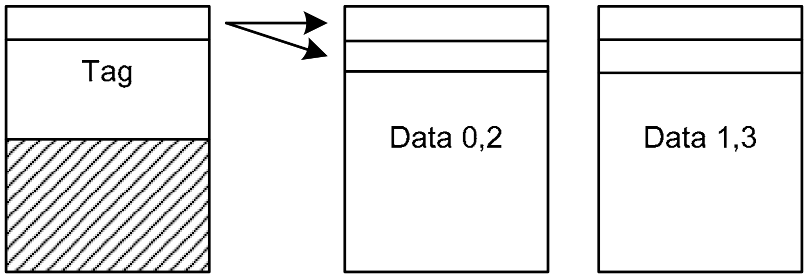


Figure 19 Association of tag mat indices with data mat

In case of TCC caches, bits [11:8] of the cache info register specify the mat which is used as the Address FIFO.

2.4.3 Memory map and address translation

Figure 20 shows the virtual and physical address spaces of the Smart Memories system. Both address spaces are 4GB and are divided into 512MB segments. Processors issue requests in the virtual space, which are then translated into physical addresses for use both inside and outside of the Tile.

The virtual address space is divided to three distinct sections: The first four segments, (0GB-1GB) are not used at all; processors never generate any memory accesses to this region. Segments 4-7 (1GB-2GB) are used by the instruction port to access code. Remaining segments (2GB-4GB) are used for accessing data. The TIE port only issues operations to the data address space, except for instruction pre-fetch (IPF) and instruction cache control (IHI, III) operations which are issued to instruction address space.

Physical address space is divided into four distinct sections: Segments 0 and 1 are considered unused and any access to these segments will generate an exception for the processors. Segment 2 contains all configuration registers/memories within the system. Segment 3 maps all the local memories in all the Tiles/Quads. Segments 4-15 are mapped to off-chip, main memory.

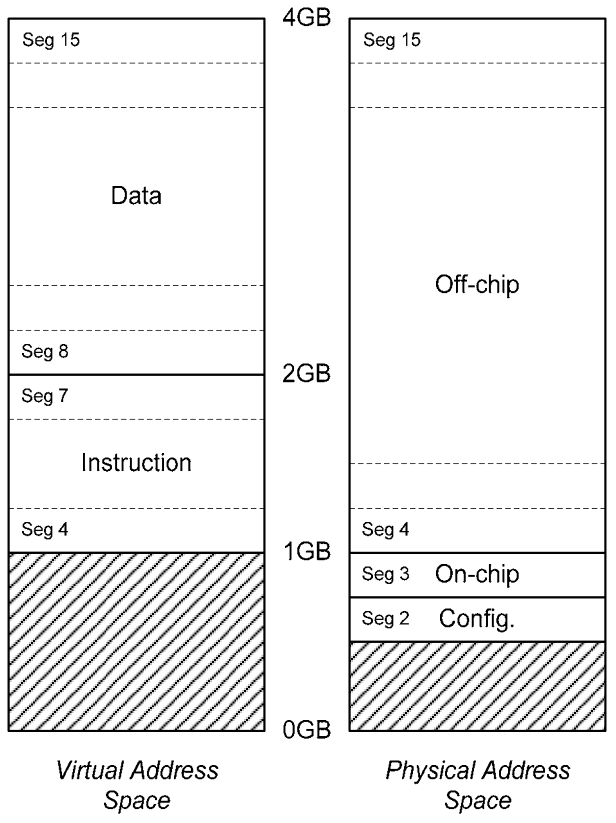


Figure 20 Virtual and physical address spaces

Translation from virtual address space to physical address space is carried out by a 12-entry segment table. This table also provides a basic memory protection mechanism and throws exceptions to processors in case of address translation error or violation of the protection. Segments can be mapped either to off-chip memory or to on-chip memory in any of the Tiles/Quads (memory mats). When mapping a segment to on-chip memory, the actual size of the segment is restricted by the segment table, since the available on-chip memory is much less than the default segment size (512MB). Segment table also indicates whether a segment is cached or not. For accesses that go to cached segments, the translated address is used to access the instruction or data caches. Note that the system does not allow caching of any segment that is mapped to on-chip memory; only segments mapped to off-chip memory can be cached. Also no segment is allowed to be mapped to configuration space. This space is solely accessed by special memory operations (RawLoad/RawStore) issued by the TIE port, which ignore the segment table and do not perform any translation. Figure 21 illustrates a segment table entry; the description for each field is as follows:

R: Read permission. If the bit is not set, read access to this segment will generate an exception.

W: Write permission. If the bit is not set, write accesses to this segment will generate and exception.

OT: On-Tile. Forces the access to go to on-Tile memory mats (ignores the bits of address that indicate Quad ID and Tile ID). Used for un-cached accesses only.

C: Cached. Indicates that the segment is cached and therefore access should go to a cache structure.

Re-map: Used for cached or off-Tile addresses. The upper four bits of the virtual address (bits 31-28) are replaced with these bits to produce physical address.

Base: Used when accessing on-chip memories only. Provides the base mat ID where segment starts.

Size: Used when accessing on-chip memories only. Size of the segment in number of mats. If the address exceeds segment boundary an exception is thrown to the issuing processor.

31	30	29	28	27 … 24	23 … 12	11 … 0
R	W	OT	C	Re-map	Base	Size

Figure 21 A segment table entry

Figure 22 illustrates the breakdown of the address bits when accessing on-chip segments. The re-map value in the segment table entry should be 3 in order to map the virtual address to on-chip memory. Quad ID, Tile ID and Mat ID uniquely identify the accessed memory mat. These values are computed by adding the 12 bits of the Base field to bits of 23-12 of virtual address. The mat index is passed to the memory mat as its address input.

31 30 29 28	27 26 25 24	23 22 21 20 19 18	17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2	1 0
Remapped bits	0	Quad id	Tile id	Mat id	Mat index	╳

Figure 22 Breakdown of address bits for accessing on-chip memories

In **Error! Reference source not found.** allocation of the configuration space (physical segment 2) to different modules in the system is shown. The configuration space covers configuration registers within memory mats, Tile LSU, crossbar and IMCN, protocol controllers and memory controllers.

2.4.4 Access translation

Instruction, data, and TIE ports of the processor issue read and write accesses to the LSU. These accesses must be translated into the appropriate mat opcode and control signals. Translation takes place according to the following criteria:

Access port (Instruction, Data or TIE)

TIE opcode (for TIE port accesses) or Read/Write signal (for instruction and data accesses)

Cached/Un-cached access (extracted from segment table)

As mentioned previously, for each instruction, data or TIE port operation issued by processors, up to three different sets of mat accesses can be generated: tag access, data access and FIFO access. Figure 24 shows the translation logic for processor’s TIE port. Translation logic for processor’s instruction and data port is the same; the only difference is that instead of the TIE opcode, a read/write signal is passed to the translation logic.

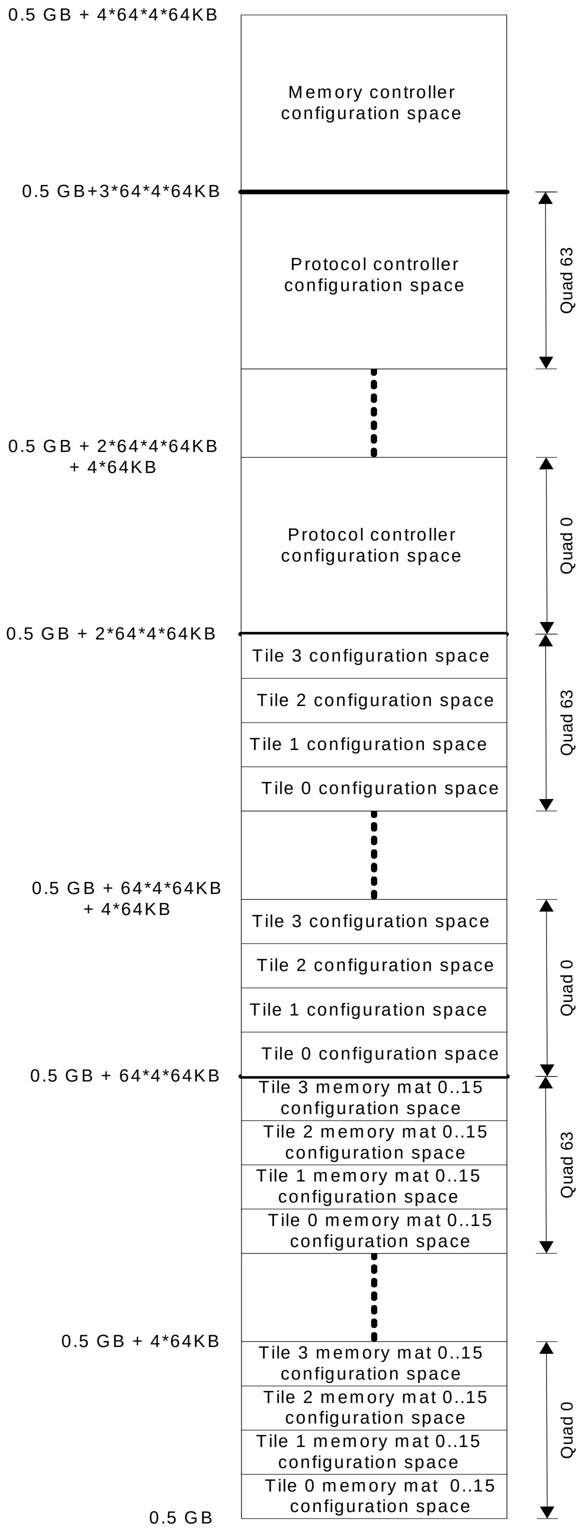

0.5 GB + 4*64*4*64KB
Memory controller configuration space
0.5 GB+3*64*4*64KB
Protocol controller configuration space
Quad 63
0.5 GB + 2*64*4*64KB + 4*64KB
Protocol controller configuration space
Quad 0
0.5 GB + 2*64*4*64KB
Tile 3 configuration space
Tile 2 configuration space
Tile 1 configuration space
Tile 0 configuration space
Quad 63
0.5 GB + 64*4*64KB + 4*64KB
Tile 3 configuration space
Tile 2 configuration space
Tile 1 configuration space
Tile 0 configuration space
Quad 0
0.5 GB + 64*4*64KB
Tile 3 memory mat 0..15 configuration space
Tile 2 memory mat 0..15 configuration space
Tile 1 memory mat 0..15 configuration space
Tile 0 memory mat 0..15 configuration space
Quad 63
0.5 GB + 4*64KB
Tile 3 memory mat 0..15 configuration space
Tile 2 memory mat 0..15 configuration space
Tile 1 memory mat 0..15 configuration space
Tile 0 memory mat 0..15 configuration space
Quad 0
0.5 GB

Figure 23 Allocation of configuration address space

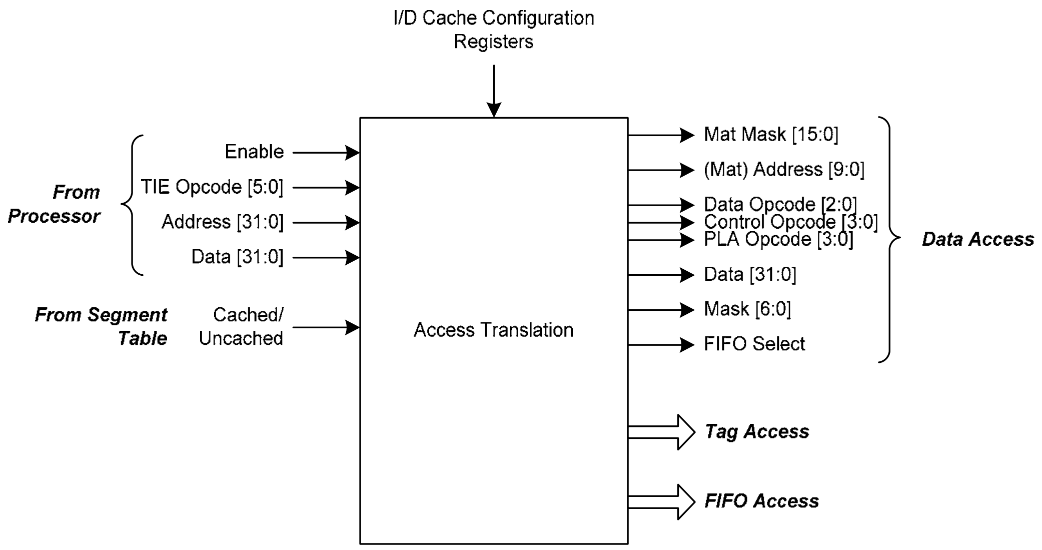


Figure 24 Access translation for the TIE port

Each mat access is comprised of the following set of signals:

Address: identifies the entry which should be accessed in the mat.

Data opcode: specifies data array operation.

Control opcode: specifies control array operation.

PLA opcode: specifies logic function of the PLA.

FIFO select: indicates whether mat is used as a FIFO (ignore the Address input and use internal pointers).

Data In: data value that is supplied to data array. It can be a write data for data mats, or tag values for tag mats (to do tag compare).

Control In: value of the control bits; for tag mats, it is the desired cache line state (Valid, Exclusive, etc). If the line is not in the desired state a cache miss will be reported.

Mask: For tag mats it indicates the control bits used in tag comparison. For data mats it is the byte mask used for performing the write in the data array.

FIFO access is generated only in the transactional mode, when mats are configured as transactional cache. Tag access is generated only if the address goes to a cached segment. Data access is always generated regardless of whether the operation is cached or un-cached. The only exception is cache control instructions issued by the TIE port (DHI, DHWBI, DHWB, DII, DIWBI, and DIWB): They only generate the tag access and no data or FIFO accesses.

A set of configuration registers inside the LSU convert the processors memory operations into appropriate set of mat signals. Each processor has its own set of registers for doing such translation. These registers are loaded with the default values upon reset, but can be written later to change the way in which a specific memory operation behaves. They usually are changed when switching to a different operational mode, for example switching to transactional mode from normal shared memory mode.

2.4.5 Communication with protocol controller

The Load/Store Unit is also responsible for communicating with the shared protocol controller located outside the Tile. The LSU uses the protocol controller to complete memory operations such as refilling caches or directly accessing a memory location outside of the Tile (in other Tiles, other Quads, or even in off-chip memory). Since there is only a single communication channel between the LSU and protocol controller, requests from instruction and data ports of the two processors have to go through an arbitration phase. The arbitration is done in round-robin fashion. Each request is tagged with a four bit sender ID signal which indicates the originating Tile ID, processor ID and port ID of the request (TIE port is considered as part of the data port for this purpose).

A small communication queue stores the requests from each port/processor to the protocol controller. After arbitration between the four queues, the winning request is sent to protocol controller, as shown in Figure 25. Protocol controller explicitly acknowledges the request after receiving and storing it locally. It is only after receiving the acknowledge signal that the LSU removes the request from its queue and moves to the next request.

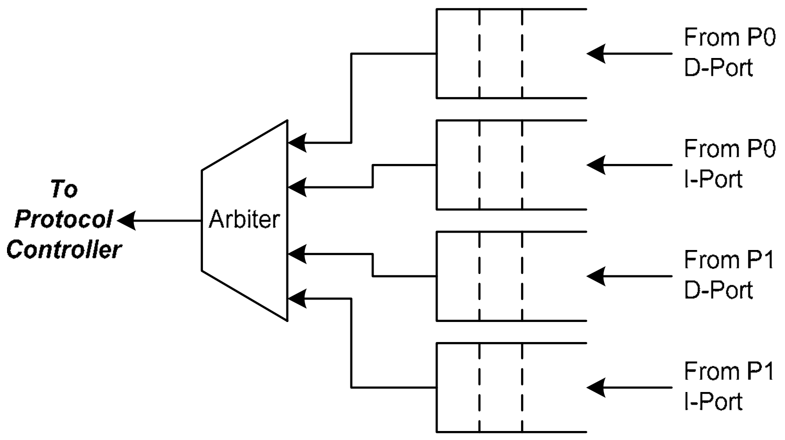


Figure 25 Communication queues to protocol controller

Table 5 lists all the communication messages between the LSU and protocol controller. Most of these require a reply back to LSU. When replying, protocol controller sends back a four bit Target ID signal indicating the processor and port.

ID	Message Type	Needs Reply?	Description
0	Off-Tile access	Yes	Direct access to a memory outside Tile
1	Cache miss	Yes	Cache miss
2	Upgrade miss	Yes	Upgrade miss (need ownership only)
3	Sync miss (un-cached)	Yes	Synchronization miss, un-cached address
4	Wakeup (un-cached)	No	Wakeup notification, un-cached address
5	Wake up (cached)	No	Wakeup notification, cached address
6	Sync miss (cached)	Yes	Synchronization miss, cached address
7	TCC FIFO full	Yes	Address FIFO in TCC cache is full
8	Hard interrupt clear	No	Acknowledgement that LSU has received hard interrupt
9	Cache control	Yes	Cache control operation (DHI, DHWB, DHWBI, DII, DIWB, DIWBI, IHI, III)
10	Pre-fetch miss	Yes	Cache miss for a pre-fetch operation

Table 5 Messages between LSU and protocol controller

2.5 Reconfigurable Protocol Controller

The protocol controller (or cache controller) is a shared configurable controller placed among the four Tiles within a Quad. It has interfaces both to memory mats within each Tile as well as to Tile Load/Store units. It is also equipped with a generic network interface to communicate with protocol controllers in other Quads or to off-chip memory controllers, and it acts as the gateway of the Quad to the rest of the system. The protocol controller provides support for the Tiles by moving data in and out of the memory mats and by implementing the desired memory access protocol.

Instead of having dedicated hardware to implement a specific memory protocol, the controller implements a set of primitive memory operations and provides a flexible means for combining and sequencing them. Handling a memory request translates to executing a sequence of primitives specific to a given protocol. This section explains the architecture of the protocol controller, describes the set of basic memory operations implemented, and shows how they are combined to implement a desired memory protocol.

2.5.1 Architecture

As mentioned, the controller executes a set of basic memory operations. These operations are divided into four main categories, as described below:

Data movements: The protocol controller transfers data between any two mats regardless of their location. It also sends and receives data over the generic network interface to other Quads or to main memory controllers.

State updates: When implementing a memory protocol such as coherence, the state information associated with the data needs to be read and updated according to the protocol rules. Since protocol controller has access to all four Tiles, it is also responsible for reading state information, using this information to decide how to proceed with the memory request and if necessary, updating the state according to the specified memory protocol.

Tracking and serialization: When implementing some memory protocols such as cache coherence, memory requests issued to same addresses need to be serialized to preserve correctness. Serialization naturally belongs in the protocol controller because all memory requests go through it. In addition, the protocol controller keeps tracking information about all outstanding memory requests from all processors.

Communication: The protocol controller has a set of interfaces to communicate with Tile processors (via LSU) and other Quads and memory controllers in the system.

In addition to these basic operations, there are a few peripherals inside the controller which are used for special operations: an interrupt unit is dedicated to assert interrupt requests for all Tile processors and to implement interrupt state machines. A configuration block provides a configuration interface for the controller such that all its internal configuration registers and memories can be accessed by an outside entity such as JTAG controller. There are also a set of eight DMA channels which can generate memory transfer requests such as strided/indexed gather/scatter operations.

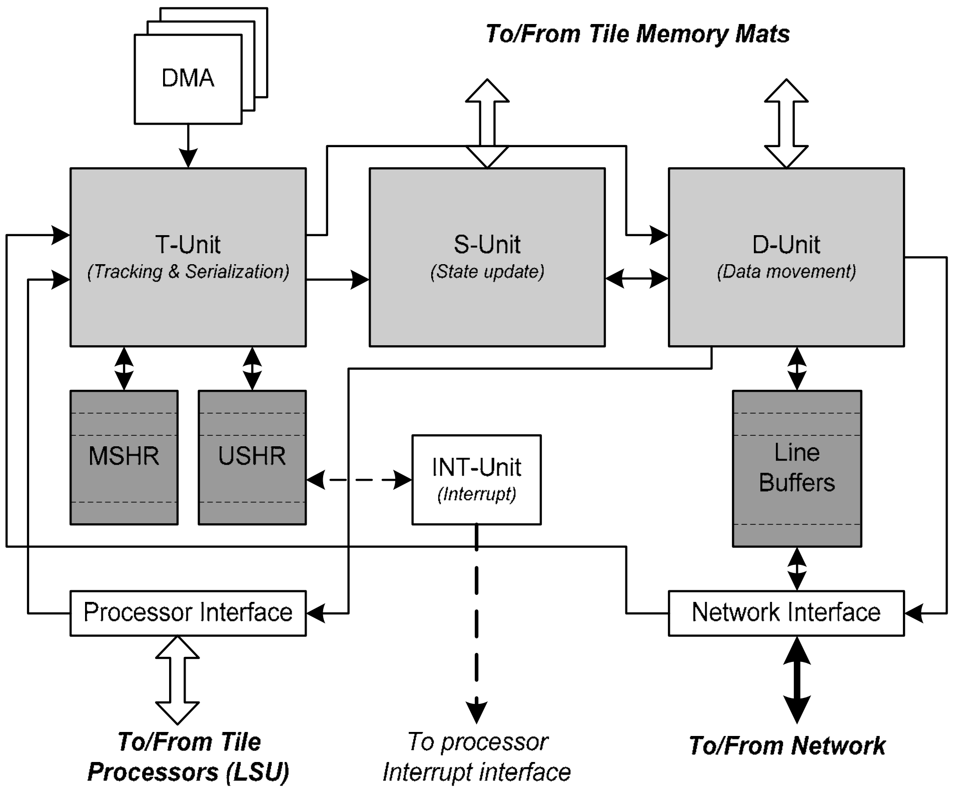


Figure 26 High level architecture of protocol controller

Figure 26 displays the high-level architecture of the protocol controller. In light gray are blocks that form the main execution core of the controller: tracking unit, state update unit and data movement unit. In dark gray are the state and data storage used by these units. These main execution units form a pipeline: they accept requests for performing operations from one another or from interfaces, execute the desired operation and pass the results to the next unit.

Internal operations of the units are controlled by a horizontal microcode which can be altered to change the behavior of the block in response to a request. The details about the internal organization and control of each unit as well as their functionality are discussed in following subsections.

2.5.2 Request tracking and serialization (T-Unit)

The Tracking Unit acts as the entry port to the execution core; it receives all request/reply messages from processors, network interface and internal DMA channels. For each request, an entry in the appropriate tracking structure is assigned and the request information is stored, after which it is passed to the next unit.

There are two separate tracking data structures managed by the tracking unit: Miss Status Holding Registers (MSHR), are used to store processor cache miss information as well as coherence requests from memory controllers. MSHR has an associative lookup port which allows the tracking unit to serialize new cache misses against already outstanding ones and enables optimizations such as request merging. Un-cached request Status Holding Registers (USHR) are separate but similar structures used to store information about a processor's direct memory requests for any locations outside of its own Tile. It also keeps information about outstanding DMA transfers generated by DMA channels.

The Tracking Unit consists of two independent parallel paths, one for handling cache miss requests and the other one for handling un-cached and miscellaneous memory requests (Figure 27). Upon receiving a new cached request, the cached request handling section (CT) allocates a new MSHR entry and looks up MSHR for other requests to the same memory block that might be already outstanding. If no collision with an already outstanding request is detected, the request is written into the MSHR during the next pipeline stage. If a request cannot be accepted due to collision with an already existing request or because MSHR structure is full, the Tracking Units returns a negative acknowledgement (Nack) and the sending party must retry the request later. When receiving a reply for an outstanding request, CT reads information of the original request from MSHR and passes it to next unit to complete processing.

The un-cached request handling part (UT) acts more or less the same way: it allocates USHR entries for incoming requests and writes them into the USHR. After receiving replies, it reads request information from the USHR ane passes it to the next stage for further processing. If the USHR becomes full, no more requests are accepted and Nacks are returned. Separate round-robin arbiters sit in front of the CT and UT paths and select the requests to be accepted by each path.

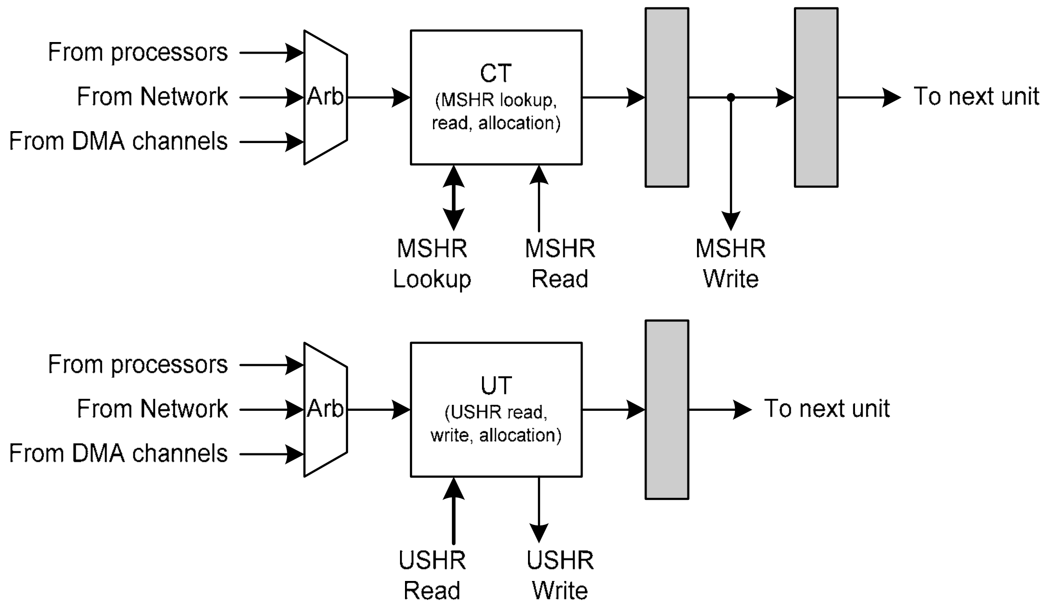


Figure 27 Tracking and serialization unit

2.5.3 State Updates (S-Unit)

State update unit is in charge of reading and updating the state information associated with data, such as cache line state, cache tags, etc. Figure 28 shows the internal organization of the S-Unit; it is a four stage pipeline with a small output queue sitting at the end. A round robin arbiter at the input selects the next request to be accepted by S-Unit.

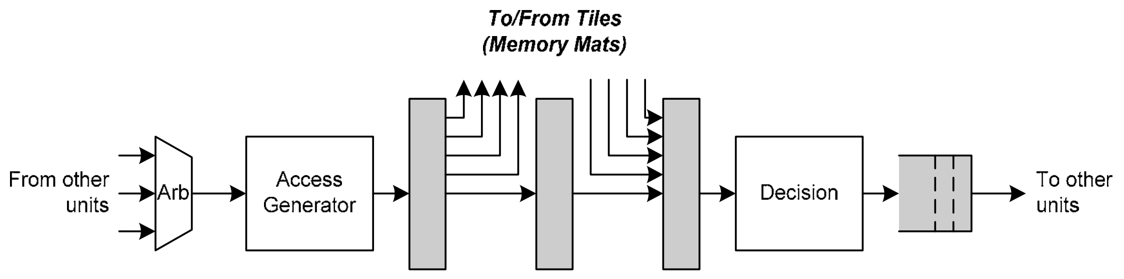


Figure 28 State update unit

The Access Generator block in the first stage of the pipeline generates all necessary signals for accessing Tile memory mats. It can generate two independent accesses to memory mats and is capable of accessing either a single Tile or all Tiles simultaneously; for example, when handling a cache miss request, it can evict a line from the requesting processor’s cache and update the state and enforce coherence invariance in other processors’ caches. Generated accesses are flopped and sent to memory mats in the next cycle. All the necessary control signals for accessing memory mats (e.g. data opcode, control opcode, PLA opcode, mask, etc.) are stored in a microcode memory inside the access generator block which is indexed by the type of the input request to S-Unit, and hence can be adjusted according to desired memory protocol.

The Decision Block at the last stage of the pipeline receives the value of all control bits (meta-data bits) read from memory mats, as well the Total Match and Data Match signals and determines the next step in processing the request. For example, when serving a cache miss request, if a copy of the line is found in another cache, a cache-to-cache transfer request is sent to D-Unit. Or if the evicted cache turns out to be modified, a write back request is generated. The decision making is performed by feeding the collected state information into a TCAM which generates the next step of processing step and identifies the unit which should receive the request. The data and mask bits inside TCAM can be altered according to any desired protocol.

A small output queue buffers requests before sending them to other units. The size of this buffer is adjusted such that it can always drain the S-Unit pipeline, preventing pipeline stalls when a memory mat access is in flight. The arbiter logic in front of the pipeline always checks the availability of buffering space in the output queue and does not accept new requests if there is not enough free entries in the queue.

2.5.4 Data Movements (D-Unit)

Figure 29 shows the internals of the Data Movement Unit. Like previous units, an arbiter first decides which request should be accepted. The Dispatch Block determines which Tiles should be accessed as part of request processing. Four data pipes associated with the four Tiles receive requests from their input queues and send the results to their output queues. A small finite state machine generates replies for processors.

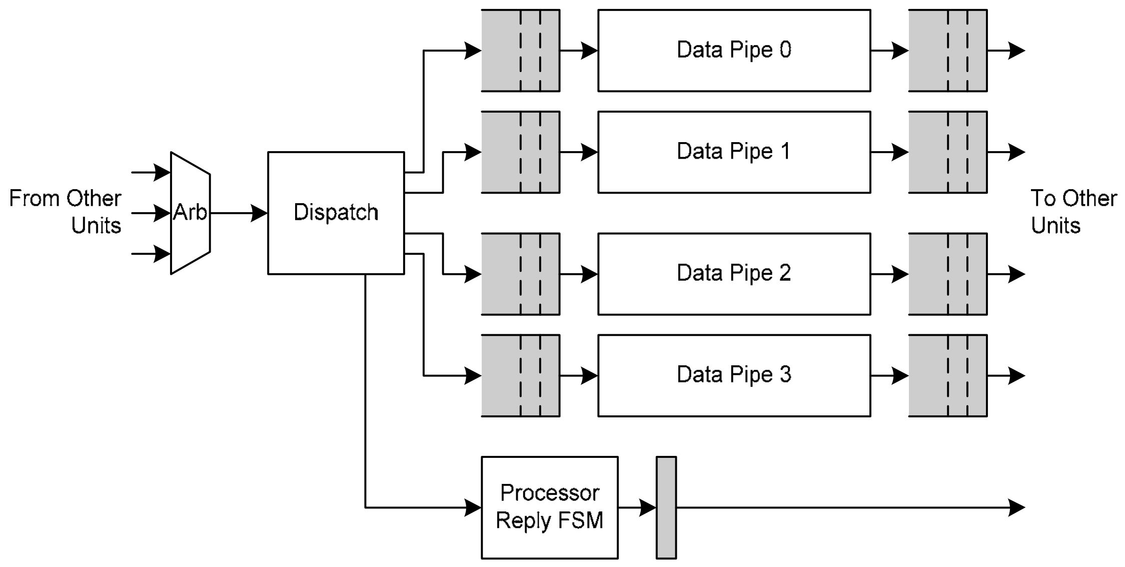


Figure 29 Data Movement Unit

The Dispatch Unit decides which Tiles are involved in a data movement operation and sends appropriate memory read/write requests to the appropriate data pipes. For example, while a simple cache refill requires writing data to memory mats of one Tile, a cache to cache transfer involves reading data from the source cache first and writing it to the destination cache with appropriate read/write scheduling. The Dispatch Unit uses an internal TCAM to determine the type and schedule of appropriate data read/writes for each data pipe and places them into data pipe input queues. It also initiates the processor reply FSM when needed.

Figure 30 is a diagram of the data pipe. It has a port to memory mats in the associated Tile as well as a read and write port to line buffer. The access generator in the first stage generates necessary control signals to read/write memory mats and line buffer. Similar to the S-Unit, all necessary signals are extracted from microcode memory within the access generator and can be changed to implement any type of access. The condition check block at the last stage receives the meta-data bits associated with the each of the accessed words and can match them with a predefined bit pattern. This allows the data pipe to generate the request for the next unit according to the extracted bit pattern. For example, when implementing fine grain locks, one of the meta-data bits in the control array of the mat is used as a Full/Empty bit. The Data pipe decides whether to reply to the processor or to send a synchronization miss message to the memory controller depending on the status of the lock bit.

A shallow output queue ensures that all the operations in the data pipe can be drained such that a memory mat access need never be stalled in flight. The dispatch unit always checks the availability of the space in the output queues of the data pipes which receive memory read/write operations and does not issue new operations unless there is enough buffering space in the output queues.

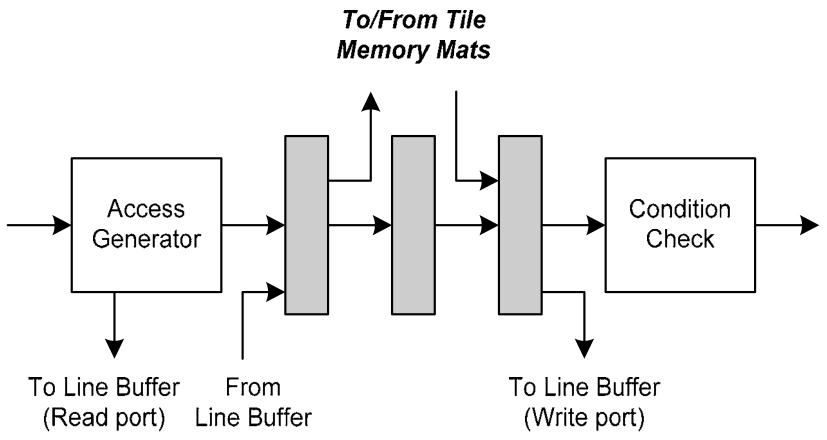

To/From Tile Memory Mats
Access Generator
Condition Check
To Line Buffer (Read port)
From Line Buffer
To Line Buffer (Write port)

Figure 30 Data pipe

2.5.5 Interfaces

The Protocol Controller has dedicated interfaces to communicate with processors as well as memory controllers and other protocol controllers. A dedicated processor interface receives requests from and returns replies to Tile processors (LSU) and chooses which request is passed to the execution core. Requests from either port of each processor are stored internally inside the processor interface and are passed to the T-Unit after winning arbitration (Figure 31). A bypass path allows a recently received request to be sent out directly if there is no other request waiting in the processor interface.

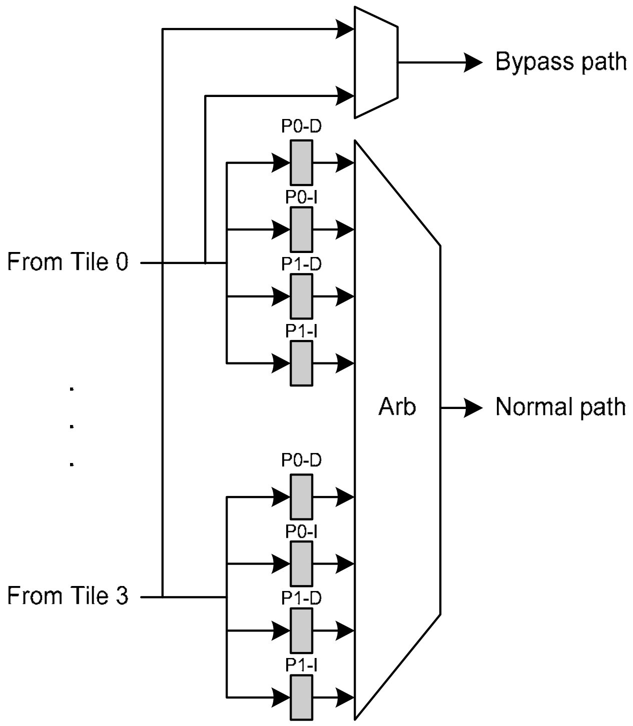


Figure 31 Processor interface logic

The network interface consists of separate receiver and transmitter blocks which operate independently. Figure 32 shows the network transmitter; a priority queue stores the requests for outbound transmissions until the transmitter is ready to send. Transmitter logic composes packet headers based on the request type and attaches the data to the header. In case of long packets, data is read from the line buffer and immediately placed in the outgoing queue. Since the IO clock rate can be configured to be slower than the system clock, a rate control queue adjusts the rate between the transmitter's internal operation and output pins.

The priority queue in front of the transmitter receives a virtual channel number along with the packet request. It considers priorities between virtual channels and selects the next request for transmission. The priority queue is sized such that it can absorb and store all active requests within the execution core. This guarantees that even when the all the outgoing links are blocked (due to back pressure, for example), all active requests in the execution core that need to send out a packet can be safely drained into the queue, releasing the execution units and preventing deadlock.

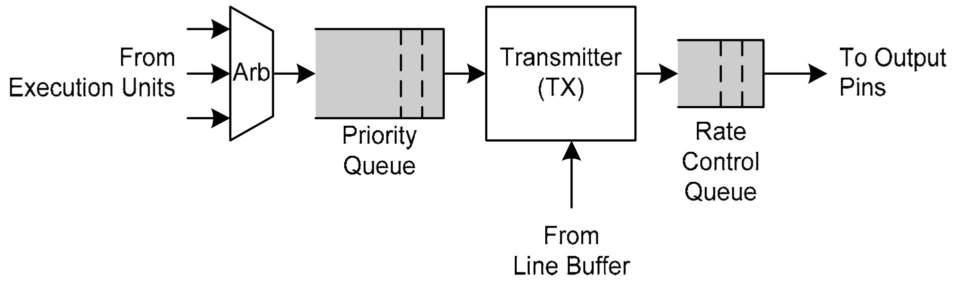


Figure 32 Network transmitter

The network receiver block is shown in Figure 33. There are eight buffers within the receiver that each store packet from a virtual channel. A decoder detects the virtual channel of the received flit and places it in the appropriate virtual channel buffer. After arbitration, header of the selected packet is decoded and a request is generated for execution core based on the packet type. In case of long packets, data words are first written into the line buffer before the request is passed to the execution core.

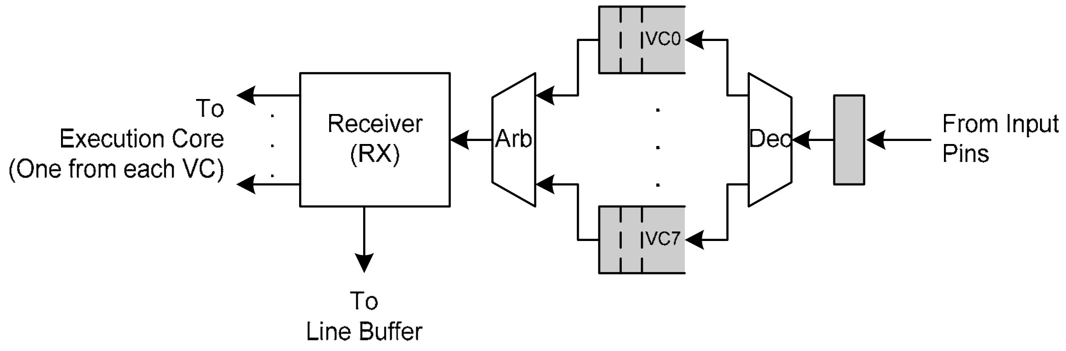


Figure 33 Network receiver

2.5.6 Peripherals

In addition to the execution units and interfaces, the protocol controller has a number of DMA channels and a small interrupt unit. Eight DMA channels generate memory transfer requests that are entered into the execution core via the tracking unit. Channels are capable of generating continuous copy requests, as well as strided and indexed gather/scatter operations. Each DMA channel is essentially a micro-coded engine which generates requests according to the loaded microcode. This makes the DMA engine a very flexible request generator which can be used by the user. For example, after completion of a transfer, DMA channels are capable of generating interrupt requests for processors who are waiting for data movement, or release lock variables on which processors are waiting. Another example is the use of DMA channels in transactions: DMA channels are used to commit the speculative modifications of the completed transaction. They extract addresses from the Address FIFO of the transactional cache, read data words from the cache itself and send them to other caches and to main memory.

A small interrupt unit connects the protocol controller to the interrupt interface of all processors. Writing into eight special registers inside this unit will generate interrupts for the corresponding processors. In addition, this interrupt unit implements a small state machine for handling a special type of interrupt called a "hard" interrupt. When a hard interrupt is issued it forces the receiving processor to come out of data access stall immediately[2] Since processors might be stalled on synchronization accesses (sync misses), a hard interrupt also has to kill any such outstanding operations before forcing processor out of stall. The state machine inside the interrupt unit sends a cancel request to the execution core, which ensures that there are no outstanding synchronization misses from a specific request.

2.5.7 Example: MESI coherence

This section describes a simple example of implementing a MESI coherence protocol inside the Quad. It is assumed that processors within a Tile share both instruction and data caches and the system consists of only one Quad. The Protocol Controller is responsible for refilling Tile caches and enforcing coherence invariance. For this purpose, a set of operations is defined for each execution unit and the whole protocol is implemented by composing these operations. First, we consider the messages the protocol controller receives from other blocks of the system. Table 6 lists all the messages along with their description.

Message Type	Source	Description
Read miss	Tile	Cache miss for a read (non-modifying) request
Write miss	Tile	Cache miss for a write (modifying) request
Upgrade	Tile	Request for cache line ownership

[2] If processor is stalled for an instruction access, such as an I-cache miss it will wait until the access is completed and then will receive the interrupt.

miss		
Refill	Memory Controller	Reply to a cache miss that was sent out previously. Brings in the cache line.

Table 6 Request/Reply messages received by protocol controller

As mentioned before, all request/replies start processing from the tracking unit (T-Unit). Table 7 lists the request types defined for T-Unit along with their operation, as well as what type of request is passed to the next unit.

Type	**Operation**	**Next Unit**	**Next Type**
CT-Read miss	- Check for collision with already outstanding request - Allocate MSHR entry - Write request information to MSHR	S-Unit	S-Read miss
CT-Write miss	Same as above	S-Unit	S-Read miss
CT-Upgrade miss	Same as above	S-Unit	S-Upgrade miss
CT-Refill	Read request information from MSHR	D-Unit	D-Refill

Table 7 T-Unit operations

As the table shows, cache misses are passed to S-Unit which searches other Tiles to find copies of the request cache line and take appropriate coherence action. Refills are given to D-Unit to write the data into the cache. Table 8 lists the operations of the S-Unit. It specifies what type of access is sent to cache of the requesting Tile as well as to caches in other Tiles and what operation is sent to the next unit based on the state information returned after cache access.

Probe access searches all the ways of a cache for a cache line and brings back the state in which the cache line is found. Degrade access also does the same, but uses the mat PLA to change the state of the cache line (if found in the cache) to shared state. It returns the old state of the cache line. Invalidate access uses PLA to change cache line state to invalid, but otherwise is the same as degrade. Evict access reads both data and meta-data bits of an index in a specified cache way, and meanwhile uses PLA to set the Reserved bit in the control array. Tag write operation is a plain write to both data and meta-data bits to fill in new tags and line state information.

Since both processors in the Tile share the first level cache, the Protocol Controller might receive cache misses from both processors. In such cases, the first cache miss brings in the cache line and refills the cache. To avoid refilling the cache for the second time, a probe access is issued to inquire the most recent state of the cache line. If the line turns out to be in the cache in appropriate state, then only the critical word requested by the processor is read or written and a reply is sent back to requesting processor.

Type	**Tile Accesses**		**Returned State**		**Next Unit**	**Next Type**
	Orig.	Other	Orig.	Other		
S-Read miss	Probe	Degrade	S, E, M	-	D-Unit	D-Critical word access
			I	S, E, M	D-Unit S-Unit	D-Cache-to-cache transfer S-Evict
			I	I	N-Unit S-Unit	N-Read miss S-Evict
S-Write miss	Probe	Invalidate	S, E, M	-	D-Unit	D-Critical word access
			I	S, E, M	D-Unit S-Unit	D-Cache-to-cache transfer S-Evict
			I	I	N-Unit S-Unit	N-Write miss S-Evict
S-Upgrade miss	Probe	Invalidate	S, E, M	-	D-Unit	D-Critical word access & update tag
			I	-	N-Unit S-Unit	N-Write miss S-Evict

S-Evict	Evict	-	M	-	D-Unit	D-Writeback
S-Evict	Evict	-	I, S, E	-	-	-
S-Tag write	Tag write	-	-	-	-	-

Table 8 S-Unit operations

Table 9 lists the D-Unit operations. The tile access column explains what accesses are sent to Tile memory mats. In some accesses, only a single word in the cache is accessed, while in others the whole cache line is read or written. According to type of the request, the dispatch block inside the D-Unit decides whether to activate one or more data pipes to complete the data transfer. In case of refill and writeback operations, the other end of the transfer is the network interface, which will write data to or read data from the line buffer.

Type	**Tile Access**		**Line Buffer**		**Next Unit**	**Next Type**
	First	Second	First	Second		
D-Critical word access	Word access	-	-	-	P-Unit	Reply
D-Critical word access & update tag	Word access	-	-	-	S-Unit P-Unit	S-Tag write Reply
D-Cache to cache transfer	Line Read	Line Write	Write	Read	S-Unit P-Unit	S-Tag write Reply
D-Writeback	Line Read	-	Write	-	N-Unit	N-Writeback
D-Refill	Line Write	-	Read	-	S-Unit P-Unit	S-Tag write reply

Table 9 D-Unit operations

Figure 34 shows the processing steps for a read miss, when the evicted cache line is not modified and a copy of the missing line is found in another Tile. In the first step, processor interface receives cache miss request and passes a CT-Read miss request to the T-Unit. After allocating MSHR and writing request information, T-Unit sends S-Read miss request to S-Unit. S-Unit searches the Tile caches, ensures that requesting cache is not already refilled (by doing a probe access) and issues a transfer request to D-Unit. D-Unit moves cache line from source Tile to destination Tile, sends reply to processor and a tag write command to S-Unit to write new tags and cache line state in the requesting cache.

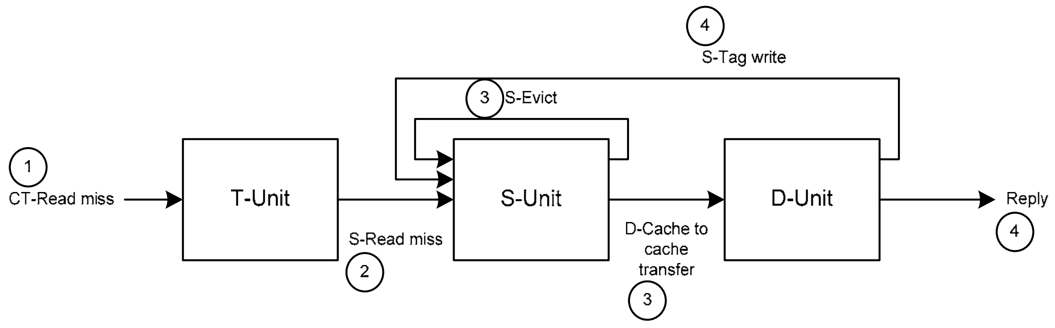


Figure 34 Processing steps for read miss

2.6 Communication Network

Smart Memories Quads are equipped with a generic network interface which enables them to communicate with each other and with peripherals such as memory controllers. This section explains this network communication in more details and provides more details about packet formats, flow control, virtual channels and their priorities, as well as the broadcast/multicast capabilities of the network.

2.6.1 Packets and flow control

In the Smart Memories network packets are divided into smaller sub-blocks called flits (flow control digit) where each flit is 78-bit wide. Each packet falls into three distinct categories: single-flit, two-flit and multi-flit. Single-flit and two-flit packets are considered short packets while any packet which has more than two flits is considered a large packet. Hence flits are divided into four different categories: Head, Tail, Body and Head_Tail. There is also a Null flit type defined which means that no flit is transmitted over the interface. Each flit carries a three bit flit type and a three bit virtual channel number in addition to the payload. Figure 35 shows the possible formats of the flit payloads.

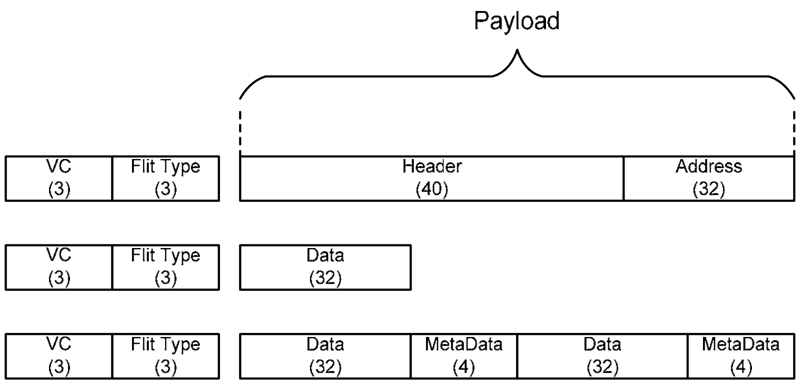


Figure 35 Flit payload formats

Packet exchange over the network is controlled by an explicit credit-based flow control mechanism; after each flit is consumed / switched by the upstream network interface, an explicit credit is sent back to the downstream network interface. Whenever the available credit is lower than the length of the packet to be sent, the packet is stalled and the interface waits for more credits before it is able to transmit again.

Figure 36 and Figure 37 show all the different packet formats exchanged over the network. Figure 36 shows the common fields in the packet headers, while Figure 37 shows the fields that differ from one packet type to other. These fields are described in the following table.

Field	Bits	Description
Broadcast mask	71-69	Used by the network switches / routers. Indicates whether and how the packet should be broadcasted to more than one destination
Destination	68-64	Destination address of the packet
Source	63-59	Source address of the packet
Message Type	58-54	Type of message
USHR Index	53-47	USHR entry which contains information of this request
MSHR Index	53-47	MSHR entry which contains information of this request
Quad ID	51-47	ID of the Quad which this request is sent on its behalf

MC MSHR Index	53-47	Memory Controller MSHR entry which contains information of this request
Opcode	46-41	Memory Opcode (load, store or TIE opcode)
Miss Type	46-44	Cache miss type (read, write, upgrade)
Line State	46-45	Cache line state (Invalid, Shared, Exclusive or Modified)
Action	46-44	Coherence action (Read, Read Exclusive, Invalidate)
Byte Mask	40-37	Byte mask for stores, used when doing direct, un-cached memory accesses
Size, Line Size	40-44	Size of the data block in the packet, in bytes (used for long packets only)
Wakeup Type	40	Type of wakeup: reader wakeup or writer wakeup
Wait Bit Adjust	40	Indicates whether there are more waiting processors on this synchronization operation
Is FIFO	39	Indicates whether the access was a remote FIFO access
FIFO Error	38	FIFO Error signal returned by the remote FIFO
FIFO Full	37	FIFO Full signal returned by the remote FIFO
Data Flag	43	Indicates that upgrade miss brings new data along with ownership
Requestor	53-32	Tile ID, processor ID and port ID of the requesting processor
I/D	32	Port ID for the request (instruction/data)
Address	31-0	Address which the request goes to (in case of long packets, the address is start address of the block)
Data	31-0	Write data for write requests, or read data for read replies

Table 10 Packet header fields

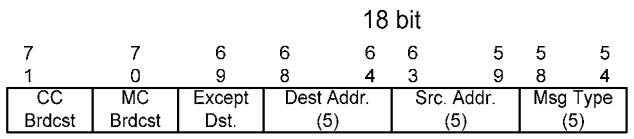


Figure 36 Common fields for headers (used by all packets)

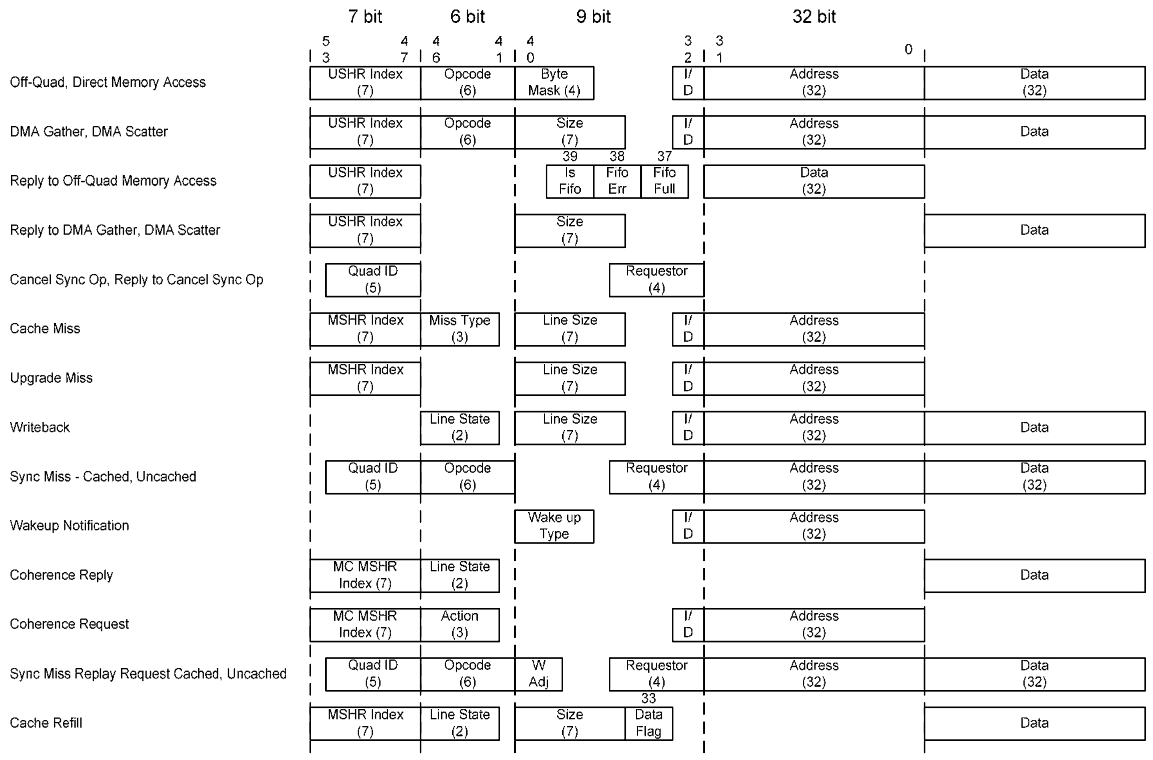


Figure 37 Fields that differ from one packet type to other. Bit 0 is where header flit ends and the next flit starts.

2.6.2 Virtual channels

Smart Memories network supports eight independent virtual channels. Virtual channel assignment for different requests/replies over the network can be configured within the Quads and memory controllers. Virtual channels support a very flexible priority scheme: for each virtual channel, an 8-bit mask indicates which other virtual channels can block it. For example, setting this mask to 8'b0000_0011 for virtual channel two indicates that it can be blocked by virtual channels zero and one, or in other words, virtual channels zero and one have priority over virtual channel two. Appropriate care is taken such that no channel can block itself.

Table 11 displays the current assignment of request/replies to virtual channels and their priorities. Note that there are some unused virtual channels within the system. VC0 is reserved for emergency messages.

VC	Message Types	Blocked by
0	Reserved	-
1	Coherence replies, cache refills, writebacks, DMA (Gather/Scatter) replies, wake up notifications, Off-Tile memory accesses, Sync miss	-
2	Coherence requests, cache misses, DMA (Gather/Scatter) requests, reply to off-tile memory accesses	VC1
3	Cancel	VC1
4	None	-
5	None	-
6	None	-
7	None	-

Table 11 Virtual channel assignments

The priority masks need to be initialized with appropriate values both in the network interfaces of Quads and memory controllers as well as intermediate switches and routers.

2.6.3 Broadcast / Multicast over network

Smart Memories network provides some basic facilities for broadcasting or multicasting packets to multiple receivers. For example, when canceling an outstanding synchronization operation, a Quad needs to broadcast the cancel message to all memory controllers. To enforce coherence, a memory controller needs to send a coherence message to all Quads except the one originating the cache miss. The broadcast/multicast features of the network allows network interfaces to send out a single request rather than generating separate requests for all desired destinations.

Three most significant bits of the packet header are used to specify broadcast/multicast. First bit, CCBroadcast, is used by memory controllers and indicates that the packet needs to be sent to all Quads. Second bit, MCBroadcast, used by Quads and indicates that the message has to be broadcasted to all memory controllers[3]. The third bit, ExceptDest, makes an exception for the node indicated by destination address. For example, if a memory controller sets these bits to value 3'b101 with the destination address set to 5'd1, the request is sent to all Quads except Quad 1. If needed, a packet can be broadcasted to all Quads and to all memory controllers by setting both CCBroadcast and MCBroadcast flags. Note that ExceptDest flag is effective only if one of the previous two flags is used.

2.7 Memory Controller

The Memory Controller is another configurable controller shared among Quads. It acts as the next level of system hierarchy above Quads, and provides access to main memory and support for memory protocols. Examples of the latter are enforcement of coherence invariance among the Quads and provision of transactional memory properties.

The Memory Controller communicates with Quads and via a generic network interface and accesses main memory through a dedicated memory interface. A Smart Memories system can be configured to have more than one memory controller. In such systems, each memory controller is in charge of a separate memory bank and memory addresses are interleaved between banks.

Since all Quads send requests to a memory controller, it naturally acts as the serialization point between them, which is important for implementing memory protocols such as coherence. Similar to a Quad's protocol controller, the Memory Controller supports a set of basic operations and implements protocols via combinations of these operations.

2.7.1 Architecture

Overall architecture of the memory controller is shown in Figure 38. Basic operational units are light gray and state storage structures are dark gray. Operational units are distinguished by the type of requests that they handle. C-Req and C-Rep units are dedicated to cache misses and coherence operations. The U-Req/Rep unit handles DMA operations and un-cached accesses to off-chip memory. The Sync Unit stores synchronization misses and replays synchronization operations whenever a wake up notification is received. Operation of each of these units is described below.

[3] Quads and memory controllers are distinguished by the MSB of their address: all memory controllers have the MSB of their ID set to one, while all Quads have the MSB of their ID set to zero.

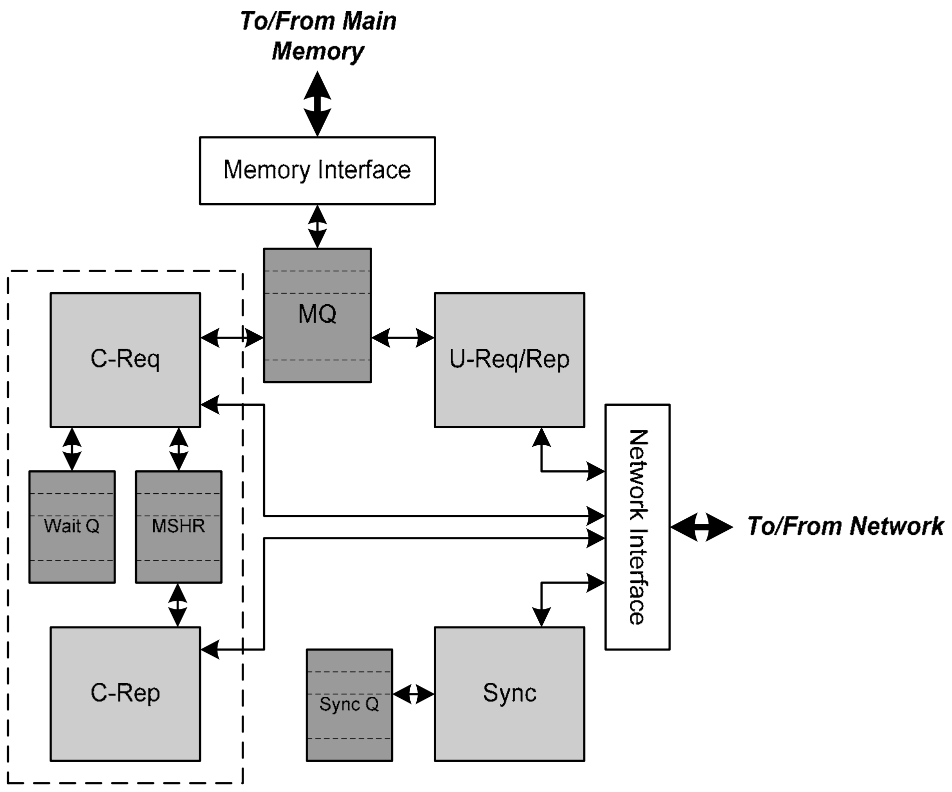


Figure 38 Architecture of memory controller

2.7.2 C-Req / C-Rep units

These two units handle cache miss and coherence operations. They basically integrate the request tracking and serialization, state monitoring and necessary data movements required for handling cache miss operations in one place. In general, memory accesses that require a form of coordination between Quads, such as cache misses in a cache coherent system or commit of transaction modifications in a transactional memory system, are handled by these two units.

The network interface delivers Quad requests to C-Req unit and Quad replies to C-Rep unit. Quad requests start their processing at C-Req unit. Similar to protocol controller, each incoming request is first checked against outstanding requests and is accepted only if there is no conflict. Outstanding request information is stored in the Miss Status Holding Register (MSHR) structure which has an associative lookup port to facilitate access based on memory block address. If no serialization is required and there is no conflicting request already outstanding, an incoming request is accepted by C-Req and is placed in MSHR. In case of a collision, a request is placed in the Wait Queue structure and is considered again when the colliding request in the MSHR completes.

When a memory request requires state information from other Quads to be collected or the state information in other Quads to be updated, C-Req unit commands the network interface to send appropriate requests to other Quads in the system, except the one that sent the original request. For example, in case of a cache miss request, caches of other Quads have to be searched to see if there is a modified copy of the cache line. Similarly, when committing the speculative modifications of a transaction, these modifications have to be broadcast to all other running transactions and hence made visible to them. Network interface has basic capability of broadcasting or multicasting packets to multiple receivers and is discussed in the next section. C-Req unit also communicates with the memory interface to initiate memory read/write operations when necessary.

The C-Rep unit collects replies from Quads and updates the MSHR structure accordingly. Replies from Quads might bring back memory blocks (e.g. cache lines) and are placed in the line buffers associated with each MSHR entry. After the last reply is received and based on the collected state information, C-Rep decides how to proceed. In cases where a memory block has to be returned to the requesting Quad (for example when replying to a cache miss), it also decides whether to send the memory block received from main memory or the one received from other Quads.

2.7.3 U-Req/Rep unit

This unit handles direct accesses to main memory. It is capable of performing single word read/write operation on the memory (un-cached memory accesses from processors) or block read/writes (DMA accesses from DMA channels). It has an interface to the Memory Queue structure and places memory read/write operations in the queue after it receives them from the network interface. After completion of the memory operation, it asks the network interface to sent back replies to the Quad that sent the original request.

2.7.4 Sync unit

As discussed in the earlier sections, Smart Memories utilizes a fine-grain synchronization protocol that allows processors to report unsuccessful synchronization attempts to memory controllers, also known as synchronization misses. When the state of the synchronization location changes, a wakeup notification is sent to memory controller and the failing request is retried on behalf of the processor. The Sync Unit is in charge of storing all the synchronization misses and attempting replay operations after wakeup notifications are received.

Information about synchronization misses is stored in the Sync Queue structure. Sync Queue is sized such that each processor in the system has its own entry.[4] When a synchronization miss is received, its information is recorded in the Sync Queue. When a wakeup notification is received for a specific address, the next processor which has an outstanding synchronization miss on that address is removed from Sync Queue and a replay request is sent to appropriate Quad to replay the synchronization operation.

[4] Each processor can have at most one synchronization miss outstanding.

The Sync Unit also handles Cancel requests received by the network interface. Cancel requests attempt to erase a synchronization miss from a specific processor if it exists in the Sync Queue. The Sync Unit invalidates the Sync Queue entry associated with the processor and sends a Cancel Reply message back to the Quad which sent the Cancel request.

2.7.5 Interfaces

The Memory Controller is equipped with a generic network interface which enables it to communicate with Quads in the system. It also has a generic memory interface and memory queue structure to perform read and write operations to memory. Network interface of the memory controller is essentially the same as the protocol controller network interface, as discussed in a previous section. It has separate transmit and receive blocks which are connected to input/output pins. It is capable of sending short and long packets and has basic broadcast capabilities which are discussed in more detail in the next section.

Memory interface is a generic 64-bit wide interface to a memory back that is operated by Memory Queue structure. When a unit needs to access main memory, it places its read/write request into the Memory Queue and the reply is returned to the issuing unit after memory operation is complete. Requests inside the queue are guaranteed to complete in the order in which they are placed in the queue and are never re-ordered with respect to each other. Block read/write operations are always broken intro 64-bit wide operations by the issuing units and are then placed inside the Memory Queue structure.

3 Transactional Coherence and Consistency

As described in Section 1 parallel programming is complicated and error-prone. The goal of the TCC project has been to develop easy-to-use and high-performance parallel systems using transactional memory (TM) technology. With TM, programmers simply declare that code blocks operating on shared data should execute as atomic and isolated transactions with respect to all other code. Concurrency control is the responsibility of the system. Hence, the focus of the TCC project has been on the development of programming models for TM-based software development and on the development of TM systems (hardware, runtime system, compiler). The basic concepts of transactional memory and TCC are described in Section 1.3, the following sections highlight the major extensions and accomplishments of the TCC effort.

3.1 Scalable TCC Architecture

The basic TCC architecture described in Section 1.3 and [21] provides support for transactional execution that performs well for small-scale, bus-based systems with 8 to 16 processors. However, given the ever-increasing transistor densities, large-scale multiprocessors with more than 16 processors on a single board or even a single chip will soon be available. As more processing elements become available, programmers should be able to use the same programming model for configurations of varying scales. Hence, TM is of long-term interest only if it scales to large-scale multiprocessors.

We have developed the first scalable, non-blocking implementation of TM [46]. Using continuous transactions, we can implement a single coherence protocol and provide non-blocking synchronization, fault isolation, and a simple to understand consistency model. The basis for this work is a directory-based implementation of the Transactional Coherence and Consistency (TCC) model that defines coherence and consistency in a shared memory system at transaction boundaries. Unlike other TM architectures, TCC detects conflict only when a transaction is ready to commit in order to guarantee livelock-freedom without intervention from user-level contention managers. It is also unique in its use of lazy data versioning which allows transactional data into the system memory only when a transaction commits. This provides a higher degree of fault isolation between common case transactions. To make TCC scalable, we used directories to implement three techniques: a) parallel commit with a two-phase protocol for concurrent transactions that involve data from separate directories; b) write-back commit that communicates addresses, but not data, between nodes and directories; c) all address and data communication for commit and conflict detection only occurs between processors that may cache shared data.

The new architecture for TCC hardware is non-blocking and implements optimistic concurrency control in scalable hardware using directories. The directory implementation reduces commit and conflict detection overheads using a two phase commit scheme for parallel commit and writeback caches. The directory also acts as a conservative filter that reduces commit and conflict detection traffic across the system. We have demonstrated that the proposed TM architecture scales efficiently to 64 processors in a distributed shared-memory (DSM) environment for both scientific and commercial workloads. Speedups with 32 processors range from 11 to 32 and for 64 processors, speedups range from 16 to 57. Commit overheads and interference between concurrent transactions are not significant bottlenecks, less than 5% of execution time on 64 processors. The organization and operation of the scalable TCC architecture were presented in the 13th International Conference on High Performance Computer Architecture.

3.2 Virtualizing Transactional Memory Hardware

For TM to become useful to programmers and achieve widespread acceptance, it is important that transactions are not limited to the physical resources of any specific hardware implementation. TM systems should guarantee correct execution even when transactions exceed scheduling quanta, overflow the capacity of hardware caches and physical memory, or include more independent nesting levels than what the hardware supports. In other words, TM systems should transparently virtualize time, space, and nesting depth. While recent application studies have shown that the majority of transactions will be short-lived and will execute quickly with reasonable hardware resources, the infrequent long-lived transactions with large data sets must also be handled correctly and transparently.

Existing HTM proposals are incomplete with respect to virtualization. None of them supports nesting depth virtualization, and most do not allow context switches or paging within a transaction (TCC, LTM, LogTM). UTM and VTM provide time and space virtualization but require complex hardware and firmware to manage overflow data structures in memory and to facilitate safe sharing among multiple processors. Since long-lived transactions are not expected to be the common case, such a complex and inflexible approach is not optimal.

In this work, we developed the first comprehensive study of TM virtualization that covers all three virtualization aspects: time, space, and nesting depth. We proposed eXtended Transactional Memory (XTM), a software-based system that builds upon virtual memory to provide complete TM virtualization without complex hardware [43]. When a transaction exceeds hardware resources, XTM evicts data to virtual memory at the granularity of pages. XTM uses private copies of overflowed pages to buffer memory updates until the transaction commits and snapshots to detect interference between transactions. On interrupts, XTM first attempts to abort a young transaction, swapping out transactional state only when unavoidable. We demonstrated that XTM allows transactions to survive cache overflows, virtual memory paging, context switches, thread migration, and extended nesting depths.

XTM can be implemented on top of any of the hardware transactional memory architectures. The combination is a hybrid TM system that provides the performance advantages of a hardware implementation without resource limitations. XTM supports transactional execution at page granularity in the same manner that page-based DSM systems provide cache coherence at page granularity. Unlike page-based DSM, XTM is a backup mechanism utilized only in the uncommon case when hardware resources are exhausted. Hence, the overheads of software-based virtualization can be tolerated without a performance impact on the common case behavior. Compared to hardware-based virtualization, XTM provides flexibility of implementation and lower cost. In the base design, XTM executes a transaction either fully in hardware (no virtualization) or fully in software through page-based virtual memory. Conflicts for overflowed transactions are tracked at page granularity. If virtualization is frequently invoked, these characteristics can lead to large overheads for virtualized transactions. To reduce the performance impact, we also developed two enhancements to the base XTM system. XTM-g allows an overflowed transaction to store data both in hardware caches and in virtual memory in order to reduce the overhead of creating private page copies. Further extension, XTM-e, allows conflict detection at cache line granularity, even for overflowed data in virtual memory, in order to reduce the frequency of rollbacks due to false sharing. XTM-g and XTM-e require limited hardware support, which is significantly simpler than the support necessary for hardware-based virtualization in VTM or UTM. XTM-g and XTM-e perform similar to hardware-based schemes like VTM, even for the most demanding applications.

Overall, this work described and analyzed the major tradeoffs in virtualization for transactional memory. Its major contributions are: a) We proposed XTM, a software-based system that is the first to virtualize time, space, and nesting depth for transactional memory. XTM builds upon virtual memory and provides transactional execution at page granularity. b) We developed two enhancements to XTM that reduce the overheads of page-based virtualization: XTM-g that allows gradual overflow of data to virtual memory and XTM-e that supports conflict detection at cache line granularity. c) We provided the first quantitative evaluation of TM virtualization schemes for a wide range of application scenarios. We demonstrated that XTM and its enhancements can match the performance of hardware virtualization schemes like VTM or TM systems that use serialization to handle resource limitation. Overall, we established that a software, page-based approach provides an attractive solution for transparent TM virtualization.

3.3 Hardware/Software Interface for Transactional Memory

Several proposed systems implement transactional memory in hardware (HTM) using different techniques for transactional state buffering and conflict detection. At the instruction set level, HTM systems provide only a couple of instructions to define transaction boundaries and handle nested transactions through flattening. While such limited semantics have been sufficient to demonstrate HTM’s performance potential using simple benchmarks, they fall short of supporting several key aspects of modern programming languages and operating systems such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions. Moreover, the current HTM semantics are insufficient to support recently proposed languages and runtime systems that build upon transactions to provide an easy-to-use concurrent programming model. For HTM systems to become useful to programmers and achieve widespread acceptance, it is critical to carefully design expressive and clean interfaces between transactional hardware and software before we delve further into HTM implementations.

In this work, we defined a comprehensive instruction set architecture (ISA) for hardware transactional memory [22, 23]. The architecture introduces three basic mechanisms: (1) two phase transaction commit, (2) support for software handlers on transaction commit, violation, and abort, and (3) closed- and open-nested transactions with independent rollback. Two-phase commit enables user-initiated code to run after a transaction is validated but before it commits in order to finalize tasks or coordinate with other modules. Software handlers allow runtime systems to assume control of transactional events to control scheduling and insert compensating actions. Closed nesting is used to create composable programs for which a conflict in an inner module does not restrict the concurrency of an outer module. Open nesting allows the execution of system code with independent atomicity and isolation from the user code that triggered it. The proposed mechanisms require a small set of ISA resources, registers and instructions, as a significant portion of their functionality is implemented through software conventions. This is analogous to function call and interrupt handling support in modern architectures, which is limited to a few special instructions (e.g., jump and link or return from interrupt), but rely heavily on well-defined software conventions.

We demonstrated that the three proposed mechanisms are sufficient to support rich functionality in programming languages and operating systems including transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. We also argue that their semantics provide a solid substrate to support future developments in TM software research. We describe practical implementations of the mechanisms that are compatible with proposed HTM architectures. Specifically, we presented the modifications necessary to properly track transactional state and detect conflicts for multiple nested transactions. Using execution-driven simulation, we evaluate I/O and conditional synchronization within transactions. Moreover, we explore performance optimizations using nested transactions.

Overall, this work is an effort to revisit concurrency support in modern instruction sets by carefully balancing software flexibility and hardware efficiency. Our specific contributions are: a) We propose the first comprehensive instruction set architecture for hardware transactional memory that introduces support for two-phase transaction commit; software handlers for commit, violation, and abort; and closed- and open-nested transactions with independent rollback. b) We demonstrate that the three proposed mechanisms provide sufficient support to implement functionality such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. No further concurrency control mechanisms are necessary for user or system code. c) We implement and quantitatively evaluate the proposed ISA. We demonstrate that nested transactions lead to 2.2*x* performance improvement for SPECjbb2000 over conventional HTM systems with flat transactions. We also demonstrate scalable performance for transactional I/O and conditional scheduling.

3.4 High Level Programming with Transactional Memory

A complete transactional environment must consider both hardware support and programming model issues. Specifically, we believe a transactional memory system should have certain key features: it should provide a *programming language* model with *implicit transactions*, *strong atomicity*, and demonstrate a scalable *multiprocessor* implementation.

To understand why this is important, let us consider the alternatives to these features:

Explicit versus implicit: Some proposals require an *explicit* step to make locations or objects part of a transaction, while other proposals make the memory operations' behavior *implicit* on the transactional state. Implicit transactions require either compiler or hardware support. Older proposals often required explicit instructions or calls to treat specific locations or objects as transactional; however, most proposals now allow existing code to run both transactionally and non-transactionally based on the context. Requiring explicit transactional operations prevents a programmer from composing existing non-transactional code to create transactions. Programmers need to create and maintain transaction-aware versions of existing non-transactional code in order to reuse it.

Weak atomicity versus strong atomicity: The atomicity criteria defines how transactional code interacts with non-transactional code. In proposals with *weak atomicity*, transactional isolation is only guaranteed between code running in transactions, which can lead to surprising and non-deterministic results if non-transactional code reads or writes data that is part of a transaction's read or write set [39]. For example, non-transactional code may read uncommitted data from the transaction's write set and non-transactional writes to the transaction's read set may not cause violations. In proposals with *strong atomicity*, non-transactional code does not see the uncommitted state of transactions and updates to shared locations by non-transactional code violate transactions, if needed, to prevent data races. From a programming model point of view, strong atomicity makes it easier to reason about the correctness of programs because transactions truly appear atomic with respect to the rest of the program. However, most software implementations of transactional memory have only guaranteed weak atomicity as a concession to performance. Recently, some hardware and hybrid proposals that support unlimited transaction sizes have also only offered weak atomicity. The problem is that programs written for one atomicity model are not guaranteed to work on the other; for a transactional program to be truly portable, it has to be written with a specific atomicity model in mind, potentially hindering its reuse on other systems.

Library versus programming language: Some proposals treat transactions simply as a library, while others integrate transactions into the syntax of the programming language. There are many issues with not properly integrating concurrency primitives with programming language semantics as shown in recent work on the Java Memory Model and threads in C and C++. Clear semantics are necessary to allow modern optimizing compilers to generate safe yet efficient code for multi-processor systems as well as perform transactional memory specific optimizations.

Uniprocessor versus multiprocessor: Some proposals require a uniprocessor implementation for correctness, while others take advantage of multiprocessor scaling. Since trends indicate a move to multiprocessors, new programming languages should make it easy to exploit these resources. In order to properly evaluate transactional memory as an abstraction to simplify parallel programming, it is important for proposals to provide a multiprocessor implementation.

In this work, we introduce the Atomos transactional programming language, which is the first to include implicit transactions, strong atomicity, and a scalable multiprocessor implementation [47]. Atomos is derived from Java, but replaces its synchronization and conditional waiting constructs with transactional alternatives. The Atomos conditional waiting proposal is tailored to allow efficient implementation with the limited transactional contexts provided by hardware transactional memory. There have been several proposals from the software transactional memory community for conditional waiting primitives that take advantage of transactional conflict detection for efficient wakeup. By allowing programmers more control to specify their conditional dependencies, Atomos allows the general ideas of these earlier proposals to be applied in both hardware and software transactional memory environments.

Atomos supports *open-nested* transactions, which we found necessary for building both scalable application programs and virtual machine implementations. Open nesting allows a nested transaction to commit before its parent transaction. This allows for parent transactions to be isolated from possible contention points in a more general way than other proposals like *early release*, which only allows a program to remove a location from its read set to avoid violations.

In this work, we make the following specific contributions: a) We introduce Atomos, the first programming language with strongly atomic transactional memory and a scalable multiprocessor implementation. b) We introduce the watch and retry statements to allow fine-grained conditional waiting, which is more scalable than other coarse-grained proposals in hardware environments with limited transactional contexts. c) We introduce the open statement to create nested transactions that commit independently from their parent. d) We introduce the concept of violation handlers to transactional memory to allow virtual machine implementations to handle expected violations without rolling back.

In our evaluation, implicit transactions and strong atomicity are supported by the Transactional Coherence and Consistency (TCC) hardware transactional memory model. The scalable implementation is built on the design of the underlying Jikes Research Virtual Machine (Jikes RVM) and Transactional Coherence and Consistency protocol. Using this environment, we evaluate the relative performance of Atomos and Java to demonstrate the value of programming with transactions. We show not only savings from the removal of lock overhead, but speedup from optimistic concurrency. While Jikes RVM and TCC are well suited to supporting Atomos, there is nothing about Atomos that fundamentally ties it to these systems. Atomos's toughest requirement on the underlying transactional memory system is strong atomicity, which lends itself more naturally toward a hardware transactional memory-based implementation. Although there has been recent research into strongly atomic software transactional memory systems, native code poses a further challenge to their use by Atomos. Typically these systems prohibit the calling of native code within transactions, significantly restricting the flexibility of the program. Atomos leverages the Jikes RVM scheduler thread architecture in its implementation of conditional waiting, but the design could be adapted to other timer-based schedulers.

4 References

1. D. A. Patterson and J. L. Hennessy, “Computer Architecture: A Quantitative Approach”, 4th edition, pp. 2-4, Morgan Kaufman, 2007.

2. M. Horowitz, W. Dally, “How Scaling Will Change Processor Architecture,” *IEEE International Solid States Circuits Conference (ISSCC) Digest of Technical Papers*, pp. 132-133, February 2004.

3. V. Agarwal et al., “Clock rate versus IPC: The end of the road for conventional microarchitectures,” in *Proceedings of the International Symposium Computer Architecture (ISCA)*, pp. 248-259, June 2000.

4. V. Srinivasan et al., “Optimizing pipelines for power and performance”, *Proceedings of the International Symposium on Microarchitecture (Micro)*, pp. 333-344, December 2002.

5. A. Hartstein, T. Puzak, “Optimum power/performance pipeline depth”, *Proceedings of the International Symposium on Microarchitecture (Micro)*, pp. 117- 125, December 2003.

6. K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 2-11, October 1996.

7. L. Hammond, B. Hubbert , M. Siu, M. Prabhu , M. Chen , and K. Olukotun, “The Stanford Hydra CMP,” *IEEE Micro*, pp. 71-84, March-April 2000.

8. M. Taylor et al., “Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams,” *Proceedings of the 31st International Symposium Computer Architecture (ISCA)*, p. 2, June 2004.

9. J. H. Ahn , W. J. Dally , B. Khailany , U. J. Kapasi , A. Das, “Evaluating the Imagine Stream Architecture,” *Proceedings of the 31st International Symposium Computer Architecture (ISCA)*, p. 14, June 2004.

10. C. Kozyrakis, D. Patterson, "Vector Vs Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks," *Proceedings of the International Symposium on Microarchitecture (Micro)*, pp. 283-293, December 2002.

11. L. Barroso et al., “Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,” *Proceedings of the International Symposium Computer Architecture (ISCA)*, pp. 282-293, June 2000.

12. P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21-29, March–April 2005.

13. D. Pham et al., “The Design and Implementation of a First-Generation CELL Processor,” *IEEE International Solid States Circuits Conference (ISSCC) Digest of Technical Papers*, pp. 184-185, February 2005.

14. R. Kalla, B. Sinharoy, and J. M. Tendler, “IBM POWER5 Chip: A Dual-Core Multithreaded Processor,” *IEEE Micro*, vol. 24, no. 2, pp. 40–47, March–April 2004.

15. T. Takayanagi et al., “A Dual-Core 64b UltraSPARC Microprocessor for Dense Server Applications,” *IEEE International Solid States Circuits Conference (ISSCC) Digest of Technical Papers*, pp. 58-59, February 2004.

16. N. Sakran et al., “The Implementation of the 65nm Dual-Core 64b Merom Processor,” *IEEE International Solid States Circuits Conference (ISSCC) Digest of Technical Papers*, pp. 106-107, February 2007.

17. B. Khailany et al., "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no. 2, pp. 35-46, Mar.-Apr. 2001.

18. W. Lee et al., “Space-time scheduling of instruction-level parallelism on a raw machine,” *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 46-57, October 1998

19. W. Thies, M. Karczmarek, and S. P. Amarasinghe, “StreamIt: A Language for Streaming Applications,” *Proceedings of the International Conference on Compiler Construction*, pp. 179-196, April 2002.

20. M. Herlihy and J.E.B. Moss, “Transactional memory: Architectural support for lock-free data structures,” *Proceedings of International Symposium Computer Architecture (ISCA)*, pp. 289-300, 1993.

21. L. Hammond et al., “Transactional Memory Coherence and Consistency,” *Proceedings of International Symposium Computer Architecture (ISCA)*, p. 102, June 2004.

22. A. McDonald et al., "Architectural Semantics for Practical Transactional Memory," *Proceedings of International Symposium Computer Architecture (ISCA)*, June 2006.

23. A. McDonald et al., "Transactional Memory: The Hardware-Software Interface," *IEEE Micro*, vol. 27, no. 1, January/February 2007.

24. W. Baek et al., “The OpenTM Transactional Application Programming Interface,” *Proceedings of International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 376-387, September 2007.

25. K. Mai et al., “Architecture and Circuit Techniques for a Reconfigurable Memory Block,” *International Solid States Circuits Conference*, February 2004.

26. J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, C. Kozyrakis, “Comparing Memory Systems for Chip Multiprocessors”, *Proceedings of International Symposium Computer Architecture (ISCA)*, June 2007.

27. J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, C. Kozyrakis, “Comparative Evaluation of Memory Models for Chip Multiprocessors”,

accepted to *ACM Transactions on Architecture and Code Optimization (TACO)*.

28. S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” *IEEE Computer*, 29(12), pp. 66–76, Dec. 1996.

29. B. Lewis and D. J. Berg, “Multithreaded Programming with Pthreads,” Prentice Hall, 1998.

30. E.L. Lusk and R.A. Overbeek, "Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors," Tech. Report No. ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

31. N. Jayasena, “Memory Hierarchy Design for Stream Computing,” *PhD thesis*, Stanford University, 2005.

32. I. Buck et al., "Brook for GPUs: Stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, August 2004, pp. 777–786.

33. K. Fatahalian et al., "Sequoia: Programming The Memory Hierarchy," *Supercomputing Conference*, November 2006.

34. F. Labonte et al., "The Stream Virtual Machine," *Proceedings of International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 267-277, September 2004.

35. P. Mattson et al., "Stream Virtual Machine and Two-Level Compilation Model for Streaming Architectures and Languages," *Proceedings of the International*

Workshop on Languages and Runtimes, in conjunction with OOPSLA'04, October 2004.

36. T. Harris et al., "Transactional Memory: An Overview," *IEEE Micro*, vol. 27, no. 3, pp. 8-29, May-June 2007.

37. M.P. Herlihy, "A methodology for implementing highly concurrent data structures," *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 197–206, March 1990.

38. K.E. Moore et al., ‘‘LogTM: Log-Based Transactional Memory,’’ *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 254-265, 2006.

39. Colin Blundell et al., ‘‘Deconstructing Transactional Semantics: The Subtleties of Atomicity,” *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

40. N. Shavit and D. Touitou, ‘‘Software Transactional Memory,’’ *Proceedings Symposium Principles of Distributed Computing (PODC)*, pp. 204-213, 1995.

41. P. Damron et al., ‘‘Hybrid Transactional Memory,’’ *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 336-346, 2006.

42. S. Kumar et al., ‘‘Hybrid Transactional Memory,’’ *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 209-220, 2006.

43. J.W. Chung et al., "Tradeoffs in Transactional Memory Virtualization," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

44. B. Saha, A. Adl-Tabatabai, and Q. Jacobson, ‘‘Architectural Support for Software Transactional Memory,’’ *Proceedings of the International Symposium on Microarchitecture (Micro)*, pp. 185-196, 2006.

45. A. Shriraman et al., “Hardware Acceleration of Software Transactional Memory,” *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

46. H. Chafi et al., "A Scalable, Non-blocking Approach to Transactional Memory," *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, February 2007.

47. B. D. Carlstrom et al., "The ATOMOS Transactional Programming Language," *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

DISTRIBUTION LIST

DTIC/OCP 8725 John J. Kingman Rd, Suite 0944 Ft Belvoir, VA 22060-6218	1 cy
AFRL/RVIL Kirtland AFB, NM 87117-5776	2 cys
Official Record Copy AFRL/RVSE/Jeffrey Scott	1 cy